Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jaka Šušteršič

Nadgradnja cevovoda zvezne postavitve s senčenjem zahtev v produkcijskem okolju

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: prof. dr. Viljan Mahnič

Ljubljana, 2017

Zahvala

Zahvaljujem se mentorju prof. dr. Viljanu Mahniču za pomoč pri izdelavi diplomskega dela. Prav tako se zahvaljujem svoji družini za vso podporo tekom celotnega študija. Hvala tudi prijateljem, ki ste me vedno nasmejali in vzpodbujali tekom magistrske študijske poti, še posebej Gašperju, Lovrotu, Nejcu in Janu. Manca, hvala ti za zapiske, podporo in najboljša štiri leta.

Jaka Šušteršič, 2017

"Feel not as though it is a sphere we live on; rather an infinite plain which has the illusion of leading yourself back to the point of origin. Once we understand that all the spheres in the sky are just large infinite plains, it will be plain to see."

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Pregled področja in metodologija **3**

2.1 Zvezna postavitev 3

2.2 Avtomatsko testiranje spletnih aplikacij 4

2.3 Omejitve obstoječih oblik avtomatskega testiranja 5

2.4 Obstoječe metode zaznavanja regresij v produkcijskem okolju 7

2.5 Senčenje zahtev 8

2.6 Metodologija 11

3 Avtomatsko zaznavanje in analiza regresij programskih izdaj **13**

3.1 Arhitektura rešitve 13

3.2 Usmerjanje spletnih zahtev 16

3.3 Replikacija podatkovne baze 18

3.4 Senčenje spletnih zahtev 22

3.5 Beleženje aplikacijskih metrik 30

3.6 Spletna aplikacija za analizo in nadzor senčenja 32

3.7 Povezava s storitvijo GitHub 41

4 Integracija rešitve s postavitvenim cevovodom ciljne aplikacije **47**

4.1 Ciljna aplikacija 47

4.2 Prilagoditev aplikacije 49

4.3 Prilagoditev cevovoda zvezne postavitve 52

5 Ovrednotenje **53**

5.1 Kriteriji vrednotenja 53

5.2 Zahtevnost integracije s ciljno aplikacijo 54
5.3 Dodatna infrastruktura 54
5.4 Količina ročnega dela 55
5.5 Transparentnost za končne uporabnike 56
5.6 Zaznani tipi regresij 56
5.7 Nivo zaupanja v pravilnost izdaje 57
5.8 Primerjava z obstoječo rešitvijo Diffy 58

6 Sklep **61**

Seznam uporabljenih kratic

kratica	angleško	slovensko
AES	advanced encryption standard	napredni enkripcijski standard
CGI	common gateway interface	skupni prehodni vmesnik
CSRF	cross site request forgery	ponarejanje spletnih zahtev
HTML	hypertext markup language	označevalni jezik za oblikovanje večpredstavnostnih dokumentov
HTTP	hypertext transfer protocol	protokol za izmenjavo večpredstavnostnih vsebin
IP	internet protocol	internetni protokol
NTP	network time protocol	omrežni časovni protokol
REST	representational state transfer	predstavitveni prenos stanja
SSH	secure shell	varna lupina
SQL	structured query language	strukturiran povpraševalni jezik

Povzetek

Naslov: Nadgradnja cevovoda zvezne postavitve s senčenjem zahtev v produkcijskem okolju

V zadnjih letih se pri razvoju spletnih aplikacij za kar najhitrejšo dostavo funkcionalnosti končnim uporabnikom vse pogosteje uporablja zvezna postavitev, katere temeljni del je postavitveni cevovod. Kljub hitrejši dostavi pa pravilno delovanje novih izdaj ostaja ključnega pomena. Zadnji koraki testiranja aplikacij, ki vključujejo zaznavanje regresij v novi izdaji, so v praksi pogosto ročni, posledično pa zamudni in manj zanesljivi. V magistrskem delu naslovimo omenjeni problem in razvijemo rešitev za avtomatsko zaznavanje regresij, ki temelji na senčenju spletnih zahtev. Naša rešitev spletne zahteve transparentno podvaja v senčeno okolje z novo izdajo in s primerjavo spletnih odgovorov iz produkcijskega ter senčenega okolja avtomatsko zaznava tako vsebinske kot tudi zmogljivostne regresije nove izdaje. Omenjeno rešitev uspešno integriramo v postavitveni cevovod izbrane spletne aplikacije in jo ovrednotimo. Uvedba naše rešitve zahteva le malo režijskega dela in dodatne infrastrukture, obenem pa omogoča preverjanje večjega števila robnih pogojev in načinov uporabe testirane aplikacije kot obstoječi tipi testiranja v postavitvenem cevovodu. Poleg tega naša rešitev za razliko od edine nam znane primerljive rešitve Diffy, ki podpira samo senčenje varnih zahtev, omogoča senčenje tudi nevarnih spletnih zahtev.

Ključne besede

testiranje programske opreme, zvezna postavitev, postavitveni cevovod, senčenje zahtev

Abstract

Title: Enhancing the continuous deployment pipeline by shadowing production requests

In recent years, web application development has seen an increase in utilization of continuous delivery principles, ensuring rapid delivery of functionality to end users. However, despite faster releases, correctness of the released software remains crucial. In practice, the last steps of testing a new release involve detection of regressions from the previous release, which are performed manually, and are thus time-consuming and unreliable. In this thesis, we address this problem and develop a solution for automated regression detection based on shadowing of production requests. Our solution transparently duplicates web requests into a shadow environment used by the new release, and compares responses from production and shadow environments in order to detect content and performance regressions. We integrate our solution with the deployment pipeline of an existing web application and evaluate it. We demonstrate that introduction of our solution to deployment pipelines of existing applications requires little overhead and additional infrastructure, while at the same time enabling more thorough testing of numerous boundary cases and use cases that the existing types of testing in the deployment pipelines do not cover. To the best of our knowledge, unlike the only other comparable alternative Diffy, our solution enables shadowing of both safe as well as unsafe web requests.

Keywords

software testing, continuous deployment, deployment pipeline, request shadowing

Poglavje 1

Uvod

V zadnjih letih se pri razvoju spletnih aplikacij za kar najhitrejšo dostavo novih funkcionalnosti končnim uporabnikom vse pogosteje uporablja zvezna postavitev (angl. *continuous deployment*), katere temeljni del predstavlja postavitveni cevovod (angl. *deployment pipeline*) [18]. Postavitveni cevovod modelira avtomatsko izvajanje vseh korakov od oddaje izvorne kode v repozitorij programske kode do njene izdaje v produkcijskem okolju. Čas, potreben za dostavo novih funkcionalnosti končnim uporabnikom spletne aplikacije, se posledično zmanjša, kar omogoča hitrejše prilagajanje aplikacije novim poslovnim zahtevam [31]. Pogostejše izdaje imajo pozitiven vpliv tudi na razvijalce. Ker se nove funkcionalnosti lahko izdajajo takoj, ko so pripravljene, namesto v za to predvidenih ciklih, so razvijalci manj obremenjeni s časovnimi roki [27].

Kljub hitrejši in bolj pogosti dostavi funkcionalnosti pa pravilno delovanje izdanih rešitev ostaja ključnega pomena, zaradi česar ima testiranje programske opreme v postavitvenem cevovodu pomembno vlogo [19]. Uveljavljene oblike testiranja, na primer testi enot (angl. *unit tests*) in integracijski testi (angl. *integration tests*), običajno zadoščajo zgolj za preverjanje robnih pogojev, ki si jih zamisli razvijalec, zato pogosto ne zaznajo vseh regresij (angl. *regression*) nove izdaje. Regresija je nepredvidena in neželena sprememba v delovanju dane spletne aplikacije, ki nastane pri prehodu iz prejšnje izdaje aplikacije na novo, in sicer kot posledica spremembe izvorne kode ali konfiguracije [30]. Glede na to, ali se regresija odraža v spletnih odgovorih ali v zmanjšani zmogljivosti spletne aplikacije ločimo *vsebinske* in *zmogljivostne* regresije. Zaznavanje regresij in spremljanje delovanja izdaje v praksi običajno izvajamo ročno [23], v okviru zadnjih korakov testiranja izdaje. Zaradi slednjega so ti koraki navadno počasni, manj zanesljivi in težko ponovljivi. Celovita orodja, ki bi razvijalcem spletnih aplikacij omogočala spremljanje njihovega delovanja in avtomatsko zaznavanje regresij v novih izdajah, trenutno ne obstajajo.

Za zaznavanje regresij novih izdaj lahko uporabimo zahteve iz produkcijskega okolja

aplikacij. Najpogosteje se uporabljata metodi modro-zelenih postavitev (angl. *blue-green deployments*) in kanarskih izdaj (angl. *canary releases*). Obe sta zamudni, saj od razvijalca zahtevata ročno pregledovanje delovanja izdaj, prav tako pa sta namenjeni zgolj zaznavanju in obvladovanju obstoječih napak v produkcijskem okolju, ne pa tudi njihovemu preprečevanju. Obe pomanjkljivosti rešuje zaznavanje regresij, ki temelji na senčenju produkcijskih zahtev (angl. *request shadowing*) [27]. Pri slednjem regresije zaznavamo s primerjavo odgovorov produkcijske in testirane izdaje na enako spletno zahtevo, in jih lahko odkrijemo še pred postavitvijo nove izdaje v produkcijsko okolje [35, 39].

V pričujočem delu implementiramo rešitev za avtomatsko zaznavanje regresij novih izdaj, ki temelji na senčenju spletnih zahtev. Osredotočimo se predvsem na celovitost rešitve in varnost njene uporabe v produkcijskem okolju. Čeprav naša rešitev temelji na preprostih konceptih, zahteva implementacijo ali nadgradnjo številnih programskih modulov.

Uporaba naše rešitve razvijalcem spletnih aplikacij omogoča avtomatsko preverjanje velikega števila robnih pogojev in primerov uporabe ciljne aplikacije, ki jih obstoječi načini testiranja v postavitvenih cevovodih ne pokrivajo. Uporaba produkcijskih podatkov v postopku testiranja in avtomatiziran postopek zaznavanja in analize regresij pa vnašata višjo stopnjo zaupanja v pravilnost novih izdaj s strani razvijalske ekipe.

Našo rešitev je možno uporabljati za zaznavo regresij poljubne ciljne spletne aplikacije, ki uporablja sistem za upravljanje podatkovnih baz PostgreSQL. Za praktičen prikaz smo jo integrirali s postavitvenim cevovodom obstoječe aplikacije, izdelane z ogrodjem Ruby on Rails.

Struktura preostanka pričujočega besedila je sledeča: v poglavju 2 pregledamo področje našega dela in predstavimo metodologijo; razložimo osnovne pojme testiranja spletnih aplikacij v sklopu postavitvenega cevovoda in pristope za zaznavanje regresij novih izdaj aplikacij, ki temeljijo na podatkih iz produkcijskega okolja. V poglavju 3 podamo arhitekturo naše rešitve za avtomatsko zaznavanje regresij, ki temelji na senčenju produkcijskih spletnih zahtev, in opišemo implementacijo vseh potrebnih delov rešitve: posredniški strežnik za senčenje zahtev, vmesnega programja za sporočanje aplikacijskih metrik, aplikacije za avtomatsko izvajanje analiz in nadzorno ploščo. V poglavju 4 nato opišemo postopek integracije razvite rešitve za avtomatsko zaznavanje in analizo regresij z obstoječimi spletnimi aplikacijami in postopek integracije predstavimo na konkretni ciljni aplikaciji. V poglavju 5 ovrednotimo našo rešitev iz večih vidikov in jo primerjamo z edino nam znano primerljivo alternativo. V sklepnem delu (poglavje 6) povzamemo glavne rezultate dela in predstavimo možnosti izboljšav ter nadaljnjega dela.

Poglavje 2

Pregled področja in metodologija

2.1 Zvezna postavitev

Zvezna postavitev je princip razvoja aplikacij, ki zagotavlja avtomatizirano postavitev vsake nove izdaje aplikacije, ki uspešno pride skozi vsa preverjanja, v produkcijsko okolje. Za razliko od sorodne prakse *zvezne dostave* (angl. *continuous delivery*), kjer mora biti vsaka nova sprememba v glavni veji (angl. *master branch*) repozitorija izvorne kode zgolj pripravljena za postavitev v katerem koli trenutku in pri kateri je zadnji korak postavitve odvisen od razvijalcev [3], zvezna postavitev zahteva avtomatiziranje postavitve vsake izdaje v produkcijsko okolje. Obe praksi uvajata postavitveni cevovod, ki postopek postavitve razčleni na vse korake, ki se izvedejo od oddaje izvorne kode v repozitorij, do postavitve nove izdaje v produkcijsko okolje [18]. Postavitveni cevovod vsebuje množico korakov različnih načinov testiranja in ukazov za postavitev v poljubna izvajalna okolja. Vsaka naslednja skupina ukazov v postavitvenem cevovodu razvijalcem prinese višji nivo zaupanja v pravilnost nove izdaje, obenem pa za izvajanje zahteva več časa. S tem skrajšamo čas, potreben za prejetje prvih povratnih informacij o izdaji.

Uporaba principa zvezne postavitve ima za razvoj ciljne aplikacije številne pozitivne lastnosti. Avtomatizacija vseh korakov, ki so potrebni za integracijo, testiranje in postavitev ciljne aplikacije v produkcijsko okolje zmanjša čas, potreben za dostavo novih funkcionalnosti končnim uporabnikom. Prav tako se zaradi pogostosti izvajanja postavitev zmanjšata tveganje za napake tekom postavitve in režijsko delo s strani razvijalcev [18]. Razvijalci lahko spremembe uveljavljajo takoj, ko so končane, namesto v tedenskih ciklih, zaradi česar so manj časovno obremenjeni zaradi velikih izdaj [27]. Hitrejša do-

stava novih funkcionalnosti omogoča tudi agilnejše prilagajanje na potrebe uporabnikov in poslovne strategije.

2.2 Avtomatsko testiranje spletnih aplikacij

Preverjanje pravilnosti novih izdaj spletnih aplikacij običajno vključuje avtomatske teste, ročne teste in vzajemni pregled izvorne kode s strani razvijalcev. Rezultati omenjenih preverjanj so ključni dejavniki pri odločitvi, ali bomo posamezno programsko izdajo brez sprememb postavili v produkcijsko okolje, ali pa bomo zahtevali morebitne popravke in nadaljnji razvoj. Najpogostejše oblike testiranja se izvajajo avtomatsko, s pomočjo vnaprej pripravljene testne kode. Med avtomatske oblike testiranja uvrščamo *teste enot*, *integracijske teste*, *sprejemne teste* in *dimne teste*.

2.2.1 Testi enot

Testi enot preverjajo delovanje posameznih delov izvorne kode [26]. V primeru objektno usmerjenega programiranje so to metode posameznih razredov. Vsak test preveri točno določeno, izolirano funkcionalnost razreda. Ker testi enot praviloma ne dostopajo do podatkovne baze, se izmed vseh načinov testiranja izvajajo najhitreje. Testi enot razvijalcem omogočajo hitro (pogosto skoraj takojšnjo) povratno informacijo in odkrivanje temeljnih napak, s tem pa so bistveni za čim krajše razvojne cikle.

2.2.2 Integracijski testi

Integracijski testi preverjajo pravilnost interakcije večih delov izvorne kode [18]. V primeru objektno usmerjenega programiranja so to interakcije med več razredi. Za razliko od testov enot nam omogočajo testiranje bolj zapletenih funkcionalnosti, ki zahtevajo souporabo večih razredov in dostopajo tudi do podatkovne baze ter zunanjih storitev.

2.2.3 Sprejemni testi

Sprejemni testi (angl. *acceptance tests*) simulirajo interakcijo uporabnika s ciljno aplikacijo v testnem okolju [18]. Omogočajo nam definiranje scenarijev uporabe posameznih delov aplikacije in uporabniških interakcij, na primer vnosa podatkov v vnosni obrazec. Poleg preverjanja vsebine odgovorov HTML na posamezne zahteve, na primer pri prijavi v aplikacijo in dodajanju vnosa, omogočajo tudi preverjanje dosegljivosti posameznih podstrani spletne aplikacije.

2.2.4 Dimni testi

Dimni testi se za razliko od sprejemnih testov uporabljajo za simulacijo interakcije uporabnika s ciljno aplikacijo po postavitvi v vmesno ali produkcijsko okolje [24]. Uporaba dimnih testov nadomešča preverjanje aplikacije s strani razvijalcev po vsaki postavitvi nove izdaje, ki sicer zahteva dolgotrajno ročno obiskovanje in interakcijo s posameznimi deli aplikacije. Če izdaja uspešno prestane dimne teste, lahko sklepamo, da je bila v okolje pravilno postavljena, konfigurirana, in da deluje interakcija z zunanjimi storitvami.

2.3 Omejitve obstoječih oblik avtomatskega testiranja

Vsi predstavljeni načini avtomatskega testiranja se danes pogosto uporabljajo pri razvoju spletnih aplikacij in predstavljajo temelj za dostavo delujočih in zanesljivih izdaj. Vendar pa posamezni projekti redko uporabljajo vse predstavljene vrste avtomatskega testiranja, kar bi sicer omogočalo celovito preverjanje pravilnosti izdaj [22]. Obenem pa zaradi režijskega dela, ki ga vnaša pisanje in posodabljanje testne kode, testi pogosto ne preverjajo vseh primerov uporabe, zaradi česar v produkcijskem okolju kljub vsemu prihaja do napak. V nadaljevanju podrobneje predstavimo nekaj omejitev avtomatskega testiranja, zaradi katerih se v praksi predstavljeni načini testiranja uporabljajo le deloma, in zaradi katerih posledično potrebujemo pristope, ki za zaznavanje napak potrebujejo produkcijsko okolje.

2.3.1 Časovna obremenitev razvijalcev

Pisanje celovitega nabora avtomatskih testov za vsako na novo implementirano funkcionalnost od razvijalca zahteva veliko časa. Že zgolj preverjanje večine izvajalnih poti delov aplikacije na nivoju testov enot pogosto zahteva več vrstic izvorne kode kot implementirana funkcionalnost. Še več izvorne kode pa zahtevajo sprejemni in dimni testi, saj slednji vključujejo pripravo scenarijev za več uporabniških interakcij [18]. Obenem v praksi pogosto veliko časa posvetimo ne samo razvoju funkcionalnosti in začetnemu pisanju testov, temveč tudi njihovemu posodabljanju. Bolj kot so testi tesno sklopljeni z implementirano funkcionalnostjo, več imajo razvijalci dela z njihovim posodabljanjem tekom razvoja aplikacije [22].

Kljub vsem pozitivnim učinkom avtomatskega testiranja aplikacij torej ne moremo zanemariti časa, ki ga zahteva pisanje in posodabljanje testov, saj le-ta neposredno vpliva na količino novih funkcionalnosti, ki jih lahko implementiramo v danem časovnem okviru.

2.3.2 Rast števila izvajalnih poti

Posledica hitre dostave novih funkcionalnosti končnim uporabnikom so pogoste spremembe izvorne kode spletnih aplikacij. Za preprečevanje vnosa regresij v nove izdaje se razvijalci najpogosteje poslužujejo predvsem testov enot. Kot rečeno, slednji razvijalcem nudijo določeno stopnjo zaupanja v pravilnost nove izdaje, a obenem zahtevajo čas za pisanje in posodabljanje. Posamezni testi enot preverjajo zgolj delovanje manjših delov izvorne kode, ne nudijo pa vpogleda v pravilnost delovanja interakcije med različnimi deli kode v okviru posamezne izdaje. Obenem pa se s kompleksnostjo ciljne aplikacije število testov enot, ki bi jih potrebovali za temeljito preverjanje pravilnosti, poveča do te mere, da postane neobvladljivo [26]. Če ima metoda testiranega razreda štiri izvajalne poti (na primer pri ugnezdenem pogojnem stavku), moramo za testiranje vseh izvajalnih poti napisati štiri teste. Pri n nivojih ugnezdenih klicev metod (metoda 1 kliče metodo 2, slednja kliče metodo 3, ... metoda $n-1$ kliče metodo n) - tudi, če ima vsaka posebej zgolj štiri izvajalne poti - dejansko število izvajalnih poti naraste na 4^n. Pri $n = 5$ tako pridemo do dejanskih 1024 izvajalnih poti, za celovito testiranje pa bi potrebovali enako število testov.

2.3.3 Vhodni podatki

Testi enot, integracijski testi, sprejemni testi in dimni testi za vhodne podatke uporabljajo različne vrednosti in preverjajo različne scenarije uporabe ciljne aplikacije. Oboje si zamislijo razvijalci, in sicer tako, da z vhodnimi podatki običajno testirajo pozitivno in negativno izvajalno pot. Pozitivna izvajalna pot predstavlja predvideno delovanje funkcionalnosti s pravilnimi vhodnimi podatki, kakršne pričakujemo v večini primerov. Negativne izvajalne poti pa se izvršijo v primeru, da funkcionalnost dobi nepričakovane oziroma neveljavne vhodne podatke. Slednji vključujejo:

- napačne podatkovne tipe (npr. niz namesto številke),
- prevelike ali premajhne vrednosti (neskončnost ali nič),
- kombinacije parametrov, ki niso dovoljene zaradi poslovnih pravil,
- parametre brez vsebine (prazna vnosna polja).

Testiranje z vhodnimi podatki in scenariji, ki si jih zamisli razvijalec sam, ima tako dobre, kot tudi slabe lastnosti. Če razvijalci s testi preverjajo delovanje večjega dela funkcionalnosti spletne aplikacije, imajo v pravilnost postavitve nove izdaje več zaupanja, kot če bi bilo testov manj. Razvijalci namreč s testi preverjajo točno tiste primere uporabe, ki jih sami pričakujejo, s tem pa je manj možnosti za vnos regresij.

Kljub dejstvu, da lahko število testov v ciljni aplikaciji poljubno povečujemo, že zgolj zaradi preverjanja pravilnosti pri samo predvidenih vhodnih podatkih hitro naletimo na

mejo zmožnosti zaznavanja napak. S povečevanjem števila testov namreč lahko pride do *učinka zasičenja zaradi testiranja* (angl. *saturation effect of testing*). Slednji opisuje, da je dodajanje testov in preverjanje pravilnega delovanja funkcionalnosti aplikacij veliko lažje na začetku, ko je nabor testov majhen. Ko pa se nabor testov z rastjo kompleksnosti aplikacije povečuje, je vedno težje definirati vhodne podatke in scenarije za nove teste, ki naj bi preverjali delovanje še netestiranih delov izvorne kode. Ko dosežemo nivo zasičenja, pravilnost izdaj ni več odvisna od števila testov, saj večje število testov prinaša samo lažen občutek zaupanja v pravilnost testirane izdaje [4].

Testiranje samo z vhodnimi podatki, ki si jih zamisli razvijalec, prav tako ne omogoča odkrivanja vseh možnih regresij v novi izdaji. Poleg dejstva, da število izvajalnih poti z večanjem kompleksnosti aplikacije hitro raste, na temeljitost preverjanja pravilnosti vplivata tudi kvaliteta in količina vhodnih podatkov, ki jih razvijalec uporabi za testiranje posamezne funkcionalnosti. Testi pravilnost delovanja posamezne funkcionalnosti preverjajo z vidika razvijalcev, ki pogosto ne nastopajo v vlogi končnega uporabnika aplikacije, zaradi česar pride do razkoraka [18]. Od razvijalcev namreč ne moremo pričakovati, da bi predvideli vse možne robne primere pri uporabi vsake funkcionalnosti posebej in vseh njihovih interakcij z drugimi funkcionalnostmi. Na tem mestu omenimo besede E. W. Dijkstre, enega od pionirjev računalništva, ki v delu [8] omenja, da je "testiranje programov lahko zelo učinkovit način za prikaz obstoja napak, a obenem brezupno nezadosten način za prikaz njihovega neobstoja".

2.4 Obstoječe metode zaznavanja regresij v produkcijskem okolju

Zaradi omejitev obstoječih oblik avtomatskega testiranja, predstavljenih v razdelku 2.3, se v praksi pogosto uporabljajo metode za zaznavanje regresij novih izdaj v produkcijskem okolju aplikacij. Uporaba omenjenih metod zahteva previdnost, saj lahko privede do vpliva regresij novih izdaj na končne uporabnike. Kljub temu se jih razvijalci pogosto poslužujejo, saj predstavljajo edini način, ki celovito preverja pravilnost delovanja novih izdaj v produkcijskem okolju. Čeprav so obstoječe metode zaznavanja regresij v produkcijskem okolju danes manj znane širši skupnosti razvijalcev spletnih aplikacij, so za industrijo programske opreme zelo pomembne [9]. Za najsodobnejše implementacije testiranja novih izdaj v produkcijskem okolju so zaslužna vodilna tehnološka podjetja kot so Facebook, Amazon in Netflix. Slednja so izdala kopico poročil o uvedbi omenjenih praks [16, 17, 40].

2.4.1 Modro-zelene postavitve

Modro-zelene postavitve so način za zaznavanje in obvladovanje regresij, pri katerem ima produkcijsko okolje dve različici, od katerih se vedno uporablja le ena [18, 6]. V *modrem okolju* je postavljena trenutna produkcijska izdaja, v *zelenem okolju* pa je postavljen naslednji kandidat za produkcijsko izdajo. Privzeto se vse spletne zahteve usmerjajo v *modro okolje*, šele ob zadovoljivih rezultatih testiranja kandidata za produkcijsko izdajo z obstoječimi načini testiranja (npr. testi enot, integracijski testi) pa se jih preusmeri v *zeleno okolje*. V primeru, da po preusmeritvi spletnih zahtev zaznamo regresijo določene metrike v *zelenem okolju*, vse spletne zahteve ročno usmerimo nazaj v *modro okolje*. Morebitna regresija v novi izdaji vpliva na vse končne uporabnike od časa preusmeritve zahtev v *zeleno okolje* do zaznave regresije in posledične preusmeritve zahtev nazaj v *modro okolje*. Uvedba dveh produkcijskih sistemov znatno poveča stroške in čas, potreben za upravljanje infrastrukture [18].

2.4.2 Kanarske izdaje

Za razliko od modro-zelenih postavitev pa t. i. *kanarske izdaje* [18] uvedejo sočasno uporabo dveh različic produkcijskih izdaj, pri čemer se na novejšo izdajo preusmeri zgolj manjši del spletnih zahtev. Na ta način morebitno napačno delovanje nove izdaje občuti le manjšina končnih uporabnikov, zahteve slednjih pa so po odkritju regresije ročno preusmerjene nazaj na starejšo, delujočo izdajo. Ključna prednost kanarskih izdaj je obvladovanje tveganja, saj omogočajo testiranje na želeni skupini uporabnikov (npr. zaposlenih v podjetju) [13], kar zagotavlja nemoteno delovanje ciljne aplikacije za večino ostalih uporabnikov. Obenem pa je testiranje cenejše kot pri modro-zelenih postavitvah, saj lahko delež preusmerjenih zahtev po potrebi prilagajamo in spremljamo metrike posameznega strežnika namesto metrik dodatnega produkcijskega okolja [1].

2.5 Senčenje zahtev

Tako modro-zelene postavitve kot tudi kanarske izdaje so zamudne, saj za zaznavanje regresij zahtevajo ročno pregledovanje aplikacijskih metrik (npr. čas trajanja poizvedb v podatkovni bazi, odzivni čas aplikacije), iz katerih lahko sklepamo o pravilnosti delovanja izdaje. Prav tako pa sta oba pristopa namenjena zgolj zaznavanju in obvladovanju obstoječih napak v produkcijskem okolju, ne pa tudi njihovemu preprečevanju.

Obe pomanjkljivosti rešuje zaznavanje regresij, ki temelji na senčenju produkcijskih zahtev (angl. *request shadowing*) [27]. Pri slednjem regresije zaznavamo s primerjavo odgovorov produkcijske in testirane izdaje na enako spletno zahtevo. Pristop s senčenjem

produkcijskih zahtev omogoča avtomatsko odkrivanje zmogljivostnih in vsebinskih regresij še pred postavitvijo nove izdaje v produkcijsko okolje [35, 39].

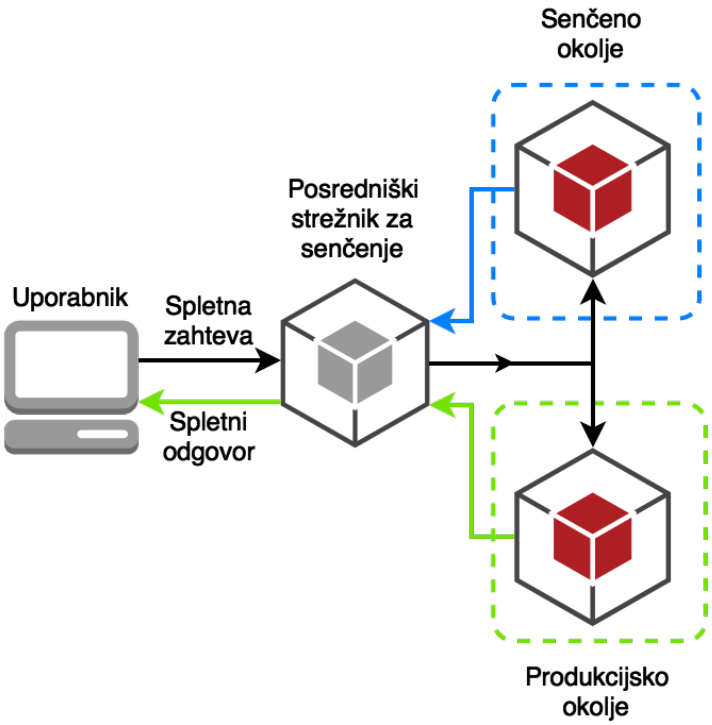
Pri senčenju se spletne zahteve, namenjene aplikaciji v produkcijskem okolju, podvojijo v dodatno aplikacijsko okolje, t. i. *senčeno* okolje. V senčenem okolju se izvaja novejša izdaja ali konfiguracija spletne aplikacije, tj. tista, ki jo testiramo pred postavitvijo v produkcijsko okolje. Aplikacijska strežnika iz obeh aplikacijskih okolij obdelata enako zahtevo, nato pa svoja odgovora posredujeta vmesni komponenti, *posredniškemu strežniku za senčenje* (angl. *shadowing proxy*). Slednji shrani oba odgovora za kasnejšo primerjavo in analizo, uporabniku pa vrne samo odgovor iz produkcijskega okolja, zaradi česar je senčenje za končnega uporabnika povsem transparentno. To pomeni, da morebitna regresija v aplikaciji, ki teče v senčenem okolju, ne bo vplivala na končnega uporabnika aplikacije. Shranjevanje odgovorov omogoča kasnejšo avtomatsko klasifikacijo razlik med shranjenimi produkcijskimi in senčenimi spletnimi odgovori na enako spletno zahtevo, na podlagi katerih lahko zaznamo zmogljivostne in vsebinske regresije. Za zaznavanje regresij lahko poleg aplikacijskih metrik (napake v delovanju, izjeme) uporabimo tudi razne sistemske (poraba procesorskega časa in pomnilnika) in poslovne metrike.

Idejno zasnovo senčenja grafično ponazarja slika 2.1. V poglavju 3 bomo videli, da je za realizacijo senčenja na implementacijskem nivoju potrebna precej bolj kompleksna arhitektura.

2.5.1 Prednosti senčenja

Ni potrebe po pisanju dodatnih testov

Ključna prednost senčenja produkcijskih zahtev v primerjavi z obstoječimi načini avtomatskega testiranja je, da razvijalcem ni potrebno implementirati testov. Senčenje namreč od razvijalca ne zahteva pisanja izvorne kode, saj namesto izmišljenih podatkov (pričakovanih vrednosti), ki jih običajno uporabljamo za odkrivanje napak pri avtomatskem testiranju, regresije v novih izdajah odkrivamo na podlagi razlik med spletnimi odgovori senčenega in produkcijskega strežnika. Pri tem odgovore produkcijskega aplikacijskega strežnika smatramo kot pravilne in pričakovane, zato morebitne spremembe v spletnih odgovorih produkcijske in testirane izdaje (v senčenem okolju) na enako spletno zahtevo lahko pomenijo regresijo. Kot bomo videli v razdelku 3.6.2, razlike v spletnih odgovorih senčenega in produkcijskega strežnika ne pomenijo nujno regresije, ampak so lahko zgolj posledica t. i. *šuma* v spletnih odgovorih.



Slika 2.1: Spletne zahteve in odgovori pri senčenju produkcijskih zahtev.

Možnost testiranja kompleksnih primerov uporabe

Pri testiranju nove izdaje aplikacije s pomočjo senčenja produkcijskih zahtev so pričakovane vrednosti, tj. vrednosti v primeru pravilnega delovanja, kar spletne zahteve in vzorci uporabniških sej iz produkcijskega okolja. Slednji poleg predvidenih parametrov spletnih zahtev in scenarijev uporabe vključujejo tudi tudi nepredvidene vhodne podatke. Zaradi kvalitetnejših testnih podatkov lahko s senčenjem dovolj velike količine produkcijskih spletnih zahtev preverimo še veliko več (predvidenih ali nepredvidenih) izvajalnih poti v spletni aplikaciji kot obstoječi načini avtomatskega testiranja, posledično pa prinašajo večje zaupanje v pravilnost nove izdaje.

Transparentnost

Pravilna in celovita implementacija senčenja nam omogoča, da učinke sprememb v izvorni kodi spletnih aplikacij testiramo na povsem transparenten način, saj regresije v testirani izdaji ne vplivajo na končne uporabnike.

2.5.2 Pomanjkljivosti senčenja

Senčenje nam avtomatsko in transparentno zaznavanje regresij v novih izdajah aplikacije omogoča na račun večje kompleksnosti, predvsem z vidika dodatne infrastrukture. V najenostavnejšem primeru namreč zahteva postavitev posredniškega strežnika za senčenje zahtev, vsaj še enega aplikacijskega strežnika in dodatne podatkovne baze.

Dodatno (t. i. senčeno) podatkovno bazo potrebujemo za zagotavljanje pravilnosti sočasne obdelave spletnih zahtev s strani produkcijskega in senčenega aplikacijskega strežnika. Za zaznavanje regresij v odgovorih na varne (angl. *safe*) spletne zahteve mora biti namreč vsebina podatkovnih baz v produkcijskem in senčenem okolju enaka, saj lahko v nasprotnem primeru med spletnima odgovoroma obeh strežnikov pride do sprememb, ki so posledica razlik v vsebini podatkovnih baz. Zaradi slednjega mora biti senčena podatkovna baza replika produkcijske, uvedba senčenja pa s tem zahteva tudi uporabo mehanizmov za upravljanje replikacije produkcijske podatkovne baze v senčeno. Replikacija je ključna tudi za zaznavanje regresij na podlagi odgovorov na nevarne (angl. *unsafe*) spletne zahteve, le da moramo v tem primeru skladnost vsebine obeh podatkovnih baz ohranjati vse do začetka obdelave zahteve, ko replikacijo prekinemo. Vse nadaljnje spremembe podatkov, ki so posledica obdelave nevarne spletne zahteve, se nato izvedejo ločeno, tj. posebej v produkcijskem ali senčenem okolju. Varne in nevarne spletne zahteve podrobneje predstavimo v razdelku 3.4.3.

2.5.3 Obstoječe rešitve

Prosto dostopnih in splošnih programskih rešitev, ki bi omogočale upravljanje podatkovne replikcije in s tem senčenje tako varnih, kot tudi nevarnih spletnih zahtev, danes ni. V primeru, da za zaznavanje regresij v spletni aplikaciji želimo uporabiti pristop s senčenjem, je trenutno najboljša alternativa odprtokodna rešitev Diffy [7], ki pa kot samostojna programska komponenta, ki ne predvideva uporabe dodatne infrastrukture, omogoča zgolj senčenje varnih zahtev. Pomanjkanje celovitih rešitev za zaznavanje regresij v spletnih aplikacijah s pomočjo senčenja produkcijskih zahtev je bila motivacija za praktični del pričujočega magistrskega dela, podrobneje predstavljenega v naslednjih poglavjih.

2.6 Metodologija

V magistrskem delu najprej definiramo visokonivojsko arhitekturo programske rešitve za avtomatsko zaznavanje regresij s pomočjo senčenja produkcijskih zahtev. Nato se lotimo razvoja posredniškega strežnika za senčenje spletnih zahtev v programskem jeziku Ruby, v okviru katerega implementiramo podvajanje spletnih zahtev iz produkcijskega aplikacij-

skega strežnika na aplikacijski strežnik v senčenem okolju. Nato vzpostavimo mehanizem za upravljanje replikacije produkcijske podatkovne baze in ga vgradimo v posredniški strežnik za senčenje zahtev. Za tem se lotimo razvoja knjižnice za sporočanje aplikacijskih metrik, ki jo bo v nadaljevanju uporabila ciljna spletna aplikacija. Razvijemo tudi spletno aplikacijo za analizo vsebine spletnih odgovorov produkcijskega in senčenega strežnika, ki med drugim za zaznavanje potencialnih regresij v novi izdaji aplikacije uporablja tudi omenjene aplikacijske metrike in pravila, definirana s strani razvijalca. Kot del omenjene spletne aplikacije razvijemo tudi nadzorno ploščo, preko katere lahko razvijalec ureja pravila, preverja spletne zahteve in odgovore produkcijskega in senčenega strežnika in preveri rezultate analiz le-teh. Za tem izberemo spletno aplikacijo z že implementiranim postavitvenim cevovodom, s katero kasneje integriramo našo rešitev za avtomatsko zaznavanje regresij. Po integraciji s ciljno spletno aplikacijo analiziramo tipe sprememb, ki se pojavljajo v spletnih odgovorih iz obeh okolij. Rezultate analiz uporabimo za definicijo pravil, na podlagi katerih lahko zaznamo regresije v ciljni spletni aplikaciji. Nazadnje našo rešitev ovrednotimo na podlagi izbranih kriterijev, kot so količina dodatne infrastrukture, količina ročnega dela, težavnost vključevanja v obstoječe rešitve in stopnja zaupanja v postavitev [33] ter jo primerjamo z edino nam znano primerljivo programsko rešitvijo za avtomatsko zaznavanje regresij, Diffy.

Poglavje 3

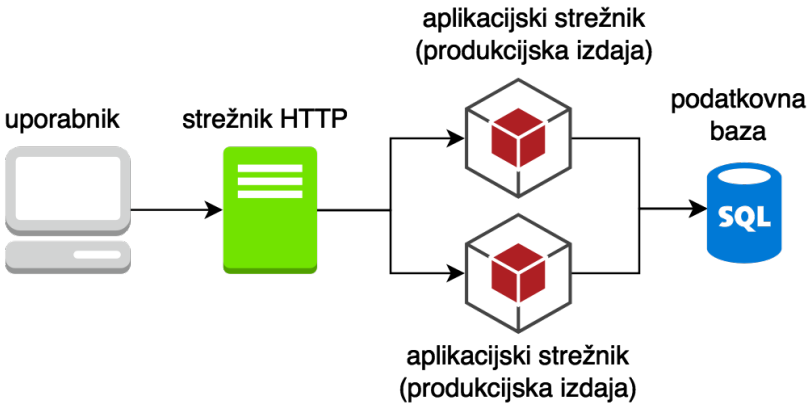
Avtomatsko zaznavanje in analiza regresij programskih izdaj

V pričujočem poglavju opišemo implementacijo vseh delov naše rešitve za avtomatsko in transparentno zaznavanje regresij posameznih izdaj ter njihovo analizo. Ključno vlogo ima posredniški strežnik za senčenje, ki izvaja senčenje spletnih zahtev v produkcijskem okolju. Poleg primerjave odgovorov produkcijskega in senčenega aplikacijskega strežnika, kakršno opisujeta deli [27] in [39], naša rešitev omogoča še analizo izbranih zmogljivostnih metrik aplikacijskega strežnika, na primer odzivnega časa in števila zahtevkov za podatkovno bazo. Rešitev smo zasnovali tako, da jo je z obstoječimi spletnimi aplikacijami kar se da enostavno integrirati.

3.1 Arhitektura rešitve

Postavitev visoko stopnjevanih (angl. *highly scalable*) spletnih aplikacij, implementiranih s pomočjo modernih spletnih ogrodij, praviloma zahteva uporabo množice aplikacijskih strežnikov in obratnega posredniškega strežnika (angl. *reverse proxy*) HTTP. Čeprav uporabniki do spletne aplikacije vedno dostopajo preko istega naslova (tj. naslova spletnega strežnika), je način ustvarjanja spletnega odgovora odvisen od vsebine, ki jo je zahteval uporabnik. Vsebino, ki jo uporabnikom streže spletni strežnik, ločimo na *statično* in *dinamično*. Med statično vsebino sodijo na primer slike in datoteke s stilskimi predlogami. Dinamično vsebino spletnih aplikacij pa predstavljajo spletne strani, ki zahtevajo obde-

lavo zahteve s strani aplikacijskega strežnika. V primeru, da uporabnik zahteva *statično vsebino*, mu neposredno odgovori spletni strežnik. Če pa uporabnik zahteva *dinamično vsebino*, spletni strežnik posreduje uporabnikovo zahtevo enemu od zalednih aplikacijskih strežnikov. Aplikacijski strežnik praviloma dostopa do podatkovne baze in na podlagi rezultatov poizvedb po podatkovni bazi ustvari spletni odgovor z dinamično vsebino. Slednjega posreduje spletnemu strežniku, ta pa končnemu uporabniku. Opisan pristop uporablja obratni posredniški strežnik, katerega glavna prednost je transparentno izpostavljanje množice aplikacijskih strežnikov na istem omrežnem naslovu in vratih. Arhitekturo postavitve spletne aplikacije z obratnim posredniškim strežnikom grafično ponazarja slika 3.1.



Slika 3.1: Tipična arhitektura postavitve z obratnim posredniškim strežnikom, ki jo uporabljajo številne visoko stopnjevane spletne aplikacije.

Uvedba posredniškega strežnika za senčenje v opisano arhitekturo postavitve vnese nekaj sprememb. Poleg štirih dodatnih entitet, in sicer posredniškega strežnika za senčenje, njegove podatkovne zbirke tipa ključ-vrednost (angl. *key-value store*) Redis za shranjevanje zahtev in odgovorov, spletne aplikacije za nadzor in analizo senčenja in replicirane podatkovne baze, se spremeni tudi interakcija med spletnim strežnikom in zalednimi aplikacijskimi strežniki. Posredniški strežnik za senčenje postavimo med spletni strežnik in izbrane zaledne aplikacijske strežnike. Kljub temu spletni strežnik vidi posredniški strežnik za senčenje zahtev zgolj kot aplikacijski strežnik, ki pa ima posebno vlogo. Arhitekturo postavitve spletnih aplikacij ob uvedbi naše rešitve grafično ponazarja slika 3.2.

Na tem mestu poudarimo, da spletno zahtevo senčimo zgolj v primeru, ko uporabnik zahteva dinamično vsebino. Samo v tem primeru namreč zaradi sprememb v izvorni kodi aplikacije lahko pride do regresije pri prehodu na novejšo verzijo spletne aplikacije.

Senčenje spletnih zahtev za statično vsebino ni smiselno, saj pri slednjih ni potrebe po komunikaciji z aplikacijskimi strežniki v zaledju.

Dinamične spletne zahteve, ki jih želimo senčiti, spletni strežnik preusmeri na posredniški strežnik za senčenje. Ta nato prejete zahteve posreduje dvema aplikacijskima strežnikoma, na katerih se izvajata različni verziji aplikacije: enemu s trenutno verzijo spletne aplikacije (slednji je v produkcijskem okolju), in enemu z novejšo verzijo (slednji je v senčenem okolju), tj. tisto verzijo, ki jo testiramo. Oba aplikacijska strežnika nato sočasno obdelata isto zahtevo in oblikujeta vsak svoj spletni odgovor. Pri tem dostopata do ločenih instanc podatkovne baze - produkcijski aplikacijski strežnik do produkcijske, senčeni aplikacijski strežnik pa do replicirane, pri čemer je vsebina replicirane podatkovne baze enaka produkcijski.

Razlogov za uporabo ločenih podatkovnih baz s strani produkcijskega in senčenega aplikacijskega strežnika je več:

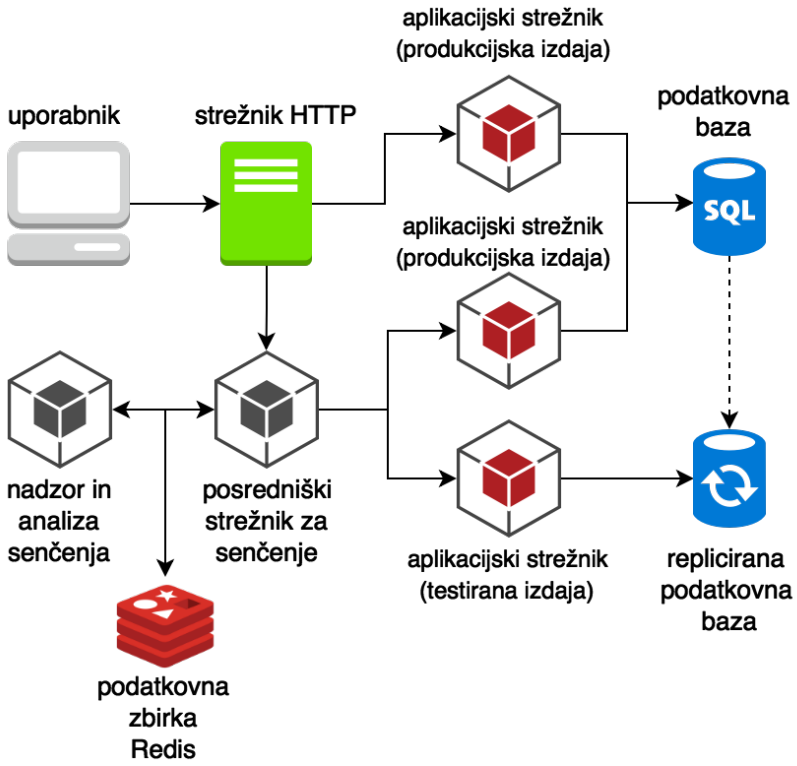
- **Pravilnost delovanja** - Zaradi sočasne obdelave iste zahteve s strani dveh aplikacijskih strežnikov bi lahko prišlo do napak v delovanju aplikacije, na primer zaradi poskusa sočasnega ustvarjanja ali brisanja istega zapisa. Napačno delovanje bi občutil tudi končni uporabnik spletne aplikacije.

- **Varnost** - Tudi če uporaba iste podatkovne baze ne bi bila problematična z vidika pravilnosti delovanja aplikacije, iz previdnosti preferiramo uporabo ločenih podatkovnih baz, saj ne želimo pomotoma vplivati na produkcijsko podatkovno bazo (npr. izbrisati kakšnega zapisa zaradi napake v aplikacijski logiki novejše verzije spletne aplikacije).

- **Učinkovitost** - Povečano obremenitev produkcijske podatkovne baze zaradi večjega števila povezav in dostopov bi lahko zaradi daljšega odzivnega časa občutili končni uporabniki spletne aplikacije.

Po obdelavi zahteve posredniški strežnik za senčenje prestreže spletna odgovora obeh aplikacijskih strežnikov in ju skupaj s pripadajočima zahtevama shrani v podatkovno zbirko Redis, kjer podatke hranimo za kasnejšo analizo in vrednotenje. Za hrambo podatkov v podatkovni zbirki tipa ključ-vrednost Redis smo se odločili, ker v primerjavi s klasičnimi sistemi za upravljanje podatkovnih baz omogoča hitrejši vnos in branje podatkov. Nazadnje posredniški strežnik za senčenje posreduje odgovor produkcijskega aplikacijskega strežnika spletnemu strežniku, slednji pa uporabniku.

Analizo odgovorov, shranjenih v podatkovni zbirki Redis, izvaja spletna aplikacija za nadzor in analizo senčenja. Ta je poleg analize spletnih odgovorov odgovorna tudi za nadzor delovanja posredniškega strežnika za senčenje.



Slika 3.2: Arhitektura postavitve ciljne spletne aplikacije ob uvedbi naše rešitve za avtomatsko zaznavanje regresij s pomočjo senčenja produkcijskih zahtev.

3.2 Usmerjanje spletnih zahtev

Za našo postavitev smo izbrali odprtokodni spletni strežnik Nginx [29], ki lahko v razširjenem načinu delovanja nastopa v vlogi obratnega posredniškega strežnika. Omogoča tudi uravnoteževanje obremenitve (angl. *load balancing*) zalednih aplikacijskih strežnikov, ki ga podrobneje predstavimo v razdelku 3.2.2. Strežnik Nginx smo izbrali, ker je bil uporabljen v postavitvi ciljne aplikacije, vendar naš posredniški strežnik za senčenje ni vezan na njegovo uporabo. Z vidika spletnega strežnika je naš posredniški strežnik za senčenje videti zgolj kot eden iz množice več aplikacijskih strežnikov, tako da ga lahko integriramo v postavitev spletne aplikacije s poljubnim spletnim strežnikom.

3.2.1 Identifikatorji zahtev

Zaradi enostavnejše analize spletnih odgovrov, ki je ena od temeljnih funkcionalnosti naše rešitve, smo na ravni strežnika Nginx implementirali označevanje spletnih zahtev z enoličnimi identifikatorji. Spletni strežnik v ta namen vsaki zahtevi v glavo HTTP doda novo polje z enoličnim identifikatorjem `X-Request-Id`. Za generiranje omenjenega identifikatorja smo uporabili kombinacijo podatkov, do katerih lahko neposredno dostopajo vse različice strežnika Nginx, in sicer: identifikator strežniškega procesa (angl. *process identifier*), časovni žig, omrežni naslov vira spletne zahteve in velikost zahteve v bajtih. Novejše različice strežnika Nginx sicer omogočajo generiranje identifikatora s pomočjo vgrajene funkcije `request_id`, vendar se zanjo nismo odločili zaradi združljivosti s starejšimi različicami strežnika. Za podporo senčenja zahtev je z vidika našega posredniškega strežnika za senčenje pomembno predvsem to, da kot del identifikatorja zahteve nastopa omrežni naslov izvorne zahteve, saj slednji omogoča sledenje uporabniškim sejam in aktivno urejanje zahtev, kot je opisano v razdelku 3.4.3.

3.2.2 Uravnoteževanje obremenitve

Večina spletnih strežnikov podpira nekaj osnovnih mehanizmov za uravnoteževanje obremenitve, tj. razporejanje spletnih zahtev med množico razpoložljivih aplikacijskih strežnikov. Na primer, strežnik Nginx podpira naslednje mehanizme:

- krožno dodeljevanje zahtev med aplikacijske strežnike (angl. *round-robin*),
- dodeljevanje zahteve aplikacijskemu strežniku z najmanj aktivnimi povezavami (angl. *least-connected*),
- dodeljevanje zahteve aplikacijskemu strežniku na podlagi zgoščene vrednosti omrežnega naslova zahteve.

Za optimalno delovanje posredniškega strežnika za senčenje moramo izbrati zadnjo možnost, kajti edino ta omogoča uporabo iste instance aplikacijskega strežnika za množico zaporednih spletnih zahtev posameznega odjemalca. Na ta način močno povečamo učinkovitost posredniškega strežnika za senčenje, saj slednji vidi sejo, ki je vzpostavljena med uporabnikom in zalednim aplikacijskim strežnikom. Nasprotno pa uravnoteževanje obremenitve s krožnim dodeljevanjem ali najmanjšim številom aktivnih povezav ne zagotavljata uporabe iste instance za več zaporednih zahtev, s tem pa otežita delo posredniškem strežniku za senčenje, saj slednji v obeh primerih nima podatkov o seji.

Uravnoteževanje obremenitve na podlagi zgolj zgoščene vrednosti omrežnega naslova zahteve ima velik vpliv na stopnjevanost ciljne spletne aplikacije. V primeru, da strežnik Nginx deluje v privzetem načinu, se na vsakega od aplikacijskih strežnikov preusmeri

enak delež zahtev. Posledično lahko delež zahtev, ki ga želimo poslati posredniškemu strežniku za senčenje, nadziramo samo na ravni spletnega strežnika, in sicer tako, da spremenimo število zalednih aplikacijskih strežnikov. Na primer, če postavimo 10 aplikacijskih strežnikov in eno instanco posredniškega strežnika za senčenje, bo slednji prejel 10 odstotkov zahtev. Če pa imamo definirane samo 4 aplikacijske strežnike, bo posredniški strežnik za senčenje prejel 25 odstotkov zahtev. Opisan način preusmerjanja zahtev na posredniški strežnik za senčenje ni optimalen, saj ne omogoča poljubne definicije deleža zahtev za senčenje pri danem številu zalednih strežnikov. V primeru, da želimo znižati delež senčenih zahtev, moramo namreč po nepotrebnem povečati število zalednih aplikacijskih strežnikov, s tem pa povečamo stroške za infrastrukturo. Nasprotno pa je za povečanje deleža senčenih zahtev potrebno zmanjšati število zalednih strežnikov, kar se odraža v slabši učinkovitosti in stopnjevanosti ciljne spletne aplikacije.

Zaradi opisane omejitve smo spletni strežnik konfigurirali tako, da uporablja *kombinirano uravnoteževanje obremenitve*. Tako uravnoteževanje za določitev aplikacijskega strežnika, h kateremu bo preusmerjena zahteva, poleg zgoščene vrednosti omrežnega naslova zahteve uporablja še uteži instanc aplikacijskih strežnikov. Poglejmo si spodnji primer konfiguracije:

```
upstream app {
    ip_hash;
    server app1.application.com weight=5;
    server app2.application.com weight=4;
    server shadow1.application.com=1;
},
```

v katerem prvemu aplikacijskemu strežniku pošljemo 50% zahtev, drugemu 40%, posredniškemu strežniku za senčenje pa 10% zahtev. Tak način uravnoteževanja obremenitve nam omogoča večji nadzor pri določanju deleža zahtev za senčenje, ne da bi pri tem neugodno vplival na stopnjevanost ciljne aplikacije ali po nepotrebnem povečal stroške infrastrukture. Obenem pa opisani pristop podpira tudi držanje seje med uporabnikom in zalednim strežnikom.

3.3 Replikacija podatkovne baze

Replikacija podatkovne baze je proces, ki omogoča vzdrževanje aktualne kopije podatkov v večih podatkovnih bazah. Uporablja se lahko za varnostno kopiranje, analizo podatkov v podatkovnih skladiščih (angl. *data warehouse*) in za doseganje višje stopnjevanosti. Kopije podatkov so lahko omejene na posamezne tabele ali na celotno podatkovno

bazo. Najbolj razširjena načina replikacije sta *nadrejeni-podrejeni* (angl. *master-slave*) in *nadrejeni-nadrejeni* (angl. *master-master*). Z uporabo načina nadrejeni-podrejeni se podatki replicirajo samo iz nadrejenega strežnika v podrejene. Zapise lahko dodaja ali ureja nadrejeni strežnik, podrejeni strežniki pa pa imajo do kopij zapisov praviloma samo bralni dostop. Opisani način omogoča večjo stopnjevanost bralnih dostopov, saj lahko uporabimo poljubno število podrejenih strežnikov. Pri uporabi načina nadrejeni-nadrejeni pa imajo za razliko od načina nadrejeni-podrejeni vsi strežniki tako bralni, kot tudi pisalni dostop do svojih kopij podatkov. Način nadrejeni-nadrejeni zato omogoča višjo stopnjevanost pisalnih dostopov, a ga je v praksi težje implementirati. Zahteva namreč obvladovanje podatkovnih konfliktov, ki nastopijo kot posledica sočasnih sprememb s strani večih strežnikov.

3.3.1 Zahteve sistema za replikacijo

Za implementacijo avtomatskega zaznavanja in analize regresij programskih izdaj smo potrebovali programsko rešitev za replikacijo produkcijske podatkovne baze v senčeno okolje. Ločena obdelava istih spletnih zahtev v senčenem in produkcijskem okolju namreč zahteva uporabo dveh podatkovnih baz z enako vsebino. Pregledali smo obstoječe rešitve za replikacijo tipa nadrejeni-podrejeni sistema za upravljanje podatkovnih baz (angl. *database management system*) PostgreSQL, v katerem smo si zastavili naslednje zahteve:

- replikacija sprememb naj traja manj kot 500 milisekund,
- postopek replikacije naj bo mogoče hitro zagnati in ustaviti,
- na podrejenem strežniku naj bo mogoč pisalni dostop do podatkov,
- replikacija naj bo z vidika nadrejenega strežnika transparentna.

Pregledali smo programske rešitve Slony-I [37], v sistem za upravljanje podatkovnih baz PostgreSQL vgrajeno funkcionalnost Streaming Replication [38] in Bucardo [2]. Vse omenjene programske rešitve zadostujejo zahtevi po hitri replikaciji sprememb podatkov. Vsem ostalim zahtevam pa zadošča samo Bucardo, ki je obenem tudi edina od rešitev, ki je transparentna z vidika nadrejenega strežnika in podpira pisalni dostop do podrejene podatkovne baze, ne da bi ji pri tem morali spremeniti vlogo v nadrejeno.

Programski rešitvi Slony-I in Streaming Replication, zaradi zagotavljanja usklajenosti (angl. *consistency*) zapisov podpirata zgolj bralni dostop do podatkov v podrejenih podatkovnih bazah, za pisalni dostop pa zahtevata spremembo vloge v nadrejeno podatkovno bazo. Sprememba vloge podrejene podatkovne baze v nadrejeno je časovno potratna, saj zahteva komunikacijo med nadrejeno in podrejeno podatkovno bazo. Poleg tega opisan pristop ni transparenten z vidika nadrejene produkcijske podatkovne baze, saj zahteva

spremembo njene vloge in posledično izpad v delovanju zaradi ponovnega zagona podatkovne baze.

3.3.2 Bucardo

Bucardo je asinhroni sistem za replikacijo podatkovnih baz sistema za upravljanje podatkovnih baz PostgreSQL, ki deluje na ravni tabel. Da zazna spremembo posameznega zapisa v dani tabeli podatkovne baze, s tem pa začne postopek replikacije, uporablja prožilce (angl. *triggers*). Pri Bucardu prožilci skrbijo za to, da se ob zaznani spremembi primarni ključi vseh spremenjenih zapisov dane tabele zapišejo v pripadajočo tabelo `delta`. Dodajanje primarnega ključa v tabelo `delta` sproži nadaljnje prožilce, ki z uporabo ukaza sistema za upravljanje podatkovnih baz PostgreSQL `NOTIFY` o spremembah obvestijo prikriti proces (angl. *daemon*) sistema Bucardo. Slednji je zadolžen za prenos sprememb v ostale (tj. replicirane) podatkovne baze, ne glede na način replikacije oziroma razmerje med podatkovnimi bazami. Replikacija lahko med poljubnim številom podatkovnih baz deluje v kombinacijah načinov nadrejeni-nadrejeni in nadrejeni-podrejeni.

Omejitve

Pri uporabi sistema za replikacijo Bucardo se moramo zavedati tudi njegovih omejitev. Bucardo ne omogoča zaznavanja sprememb sheme podatkovne baze z uporabo jezika za opis podatkov (angl. *data definition language*). Avtorji Bucarda kot razlog za to navajajo dejstvo, da v času razvoja njihovega sistema sistem za upravljanje podatkovnih baz PostgreSQL še ni podpiral prožilcev na sistemskih tabelah. Posledično je potrebno pred vsako podatkovno migracijo, ki zahteva spremembo podatkovne sheme v nadrejeni bazi, sistem ustaviti, izvesti podatkovno migracijo, nazadnje pa še prekopirati celotno podatkovno bazo. Dodatno omejitev predstavlja tudi dejstvo, da moramo vsako novo tabelo ročno dodati v konfiguracijo sistema Bucardo.

3.3.3 Začetna vzpostavitev replikacije

Za vzpostavitev replikacije podatkovne baze v načinu nadrejeni-podrejeni moramo na strežnik produkcijske podatkovne baze najprej namestiti ustrezne programske pakete sistema Bucardo, za tem pa sledi še konfiguracija sistema.

Najprej moramo vzpostaviti povezavo z lokalno instanco podatkovne baze, za kar moramo v datoteko `.pgpass` vnesti podatke za dostop (uporabniško ime in geslo). Nato v lokalni instanci podatkovne baze ustvarimo namensko podatkovno bazo in uporabnika, ki ju bo uporabljal sistem Bucardo. Za tem lahko sistem Bucardo namestimo v instanco podatkovne baze, in sicer s pomočjo ukaza `bucardo install --batch`. Obenem si za

lažje izvajanje nadaljnjih korakov postavitve v okoljske spremenljivke shranimo naslednje podatke, in sicer tako za nadrejeno kot tudi podrejeno podatkovno bazo:

- omrežni naslov in vrata,
- ime podatkovne baze,
- uporabniško ime in geslo uporabnika podatkovne baze,
- seznam tabel, ki jih želimo replicirati.

Ker želimo v podrejeni podatkovni bazi vzpostaviti enako podatkovno shemo kot v nadrejeni, v naslednjem koraku s pomočjo ukaza `pg_dump` izvozimo podatkovno shemo nadrejene baze. Omenjeni ukaz ustvari skripto SQL, ki jo zatem uvozimo v podrejeno podatkovno bazo s pomočjo ukaznega poziva sistema za upravljanje podatkovnih baz PostgreSQL, `psql`. Po vzpostavitvi podatkovne sheme v podrejeni bazi moramo konfigurirati še podrobnosti replikacije, za kar uporabimo množico ukazov, ki jih omogoča orodje ukazne vrstice `bucardo`. V ta namen najprej obe podatkovni bazi, tj. nadrejeno in podrejeno, dodamo v sistem Bucardo. V naslednjem koraku izberemo način replikacije z množico tabel in definiramo omenjene množice tabel za replikacijo iz nadrejene podatkovne baze v podrejeno. Za kopiranje obstoječih zapisov iz nadrejene v podrejeno podatkovno bazo, v prejšnjem koraku uporabimo stikalo `onetimecopy` z vrednostjo `2`. Slednje omogoča kopiranje vseh zapisov iz nadrejene podatkovne baze v podrejeno v načinu `fullcopy`, izvede pa se zgolj ob prvi uporabi. Ob koncu začetnega kopiranja se vrednost stikala nastavi na `0`, kar sproži delovanje privzetega t. i. *delnega načina replikacije* (v terminologiji sistema Bucardo se slednji imenuje *pushdelta*), ki v podrejeno podatkovno bazo vnese le tiste zapise iz nadrejene baze, ki so se spremenili od zadnje replikacije. Po zgornji konfiguraciji lahko začnemo postopek replikacije z ukazom `bucardo start`.

3.3.4 Upravljanje replikacije

Za lažje programsko upravljanje replikacije podatkovne baze smo v programskem jeziku Ruby razvili preprost vmesnik. Implementirali smo razred `BucardoController`, ki ovija sledeče ukaze orodne vrstice `bucardo` in ukaznega poziva `psql`:

- `bucardo start` za zagon storitve Bucardo,
- `bucardo stop` za ustavitev storitve Bucardo,
- `bucardo status` za prikaz stanja storitve Bucardo,
- `bucardo activate <sync_id>` za zagon replikacije,
- `bucardo deactivate <sync_id>` za ustavitev replikacije,

- `bucardo update <sync_id> onetimecopy=2` za popolno replikacijo vseh zapisov,
- `bucardo reload <sync_id>` za posodobitev nastavitev replikacije,
- `su - postgres -c 'psql -U postgres -d database_name -c "truncate public.tablename"` za brisanje vseh zapisov iz izbranih tabel.

Razred `BucardoController` navzven izpostavlja dve metodi, in sicer za ustavitev replikacije ter za popolno ponastavitev replikacije. Popolna ponastavitev replikacije nam omogoča, da iz podrejene podatkovne baze izbrišemo vse vnose, nato pa ponovno sprožimo začetek replikacije nadrejene baze. To nam koristi predvsem za usklajevanje vsebine med nadrejeno in podrejeno podatkovno bazo v primeru, da se vsebini v danem trenutku več ne ujemata.

Metodi razreda `BucardoController` z uporabo knjižnice Net-SSH [32], ki implementira odjemalca za protokol SSH, poskrbita za oddaljeno izvajanje zgoraj navedene množice ukazov na strežniku produkcijske in senčene podatkovne baze. V nadaljevanju podajamo implementacijo javne metode za popolno ponastavitev replikacije.

```
def full_sync
    Net::SSH.start(@master_ip, 'root') do |ssh|
        [:truncate_slave, :start_sync, :full_update,
        :reload, :stop, :wait, :start, :status].each do |command|
            output = ssh.exec!(COMMANDS[command])
            puts output
        end
    end
end.
```

3.4 Senčenje spletnih zahtev

V posredniškem strežniku za senčenje zahtev je bilo najprej potrebno implementirati funkcionalnost podvajanja spletnih zahtev. S pomočjo podvajanja lahko isto ali rahlo spremenjeno spletno zahtevo hkrati pošljemo večim aplikacijskim strežnikom.

Pregledali smo obstoječe rešitve za podvajanje spletnih zahtev v programskem jeziku Ruby in pri tem naleteli na več podobnih programskih knjižnic, ki za upravljanje spletnih zahtev uporabljajo knjižnico EventMachine [10]. EventMachine nudi dogodkovno vodeno upravljanje vhoda in izhoda z uporabo *reaktorskega načrtovalskega vzorca* (angl. *reactor design pattern*) in preprost vmesnik za sočasno izvajanje aplikacijske kode v obliki asinhronih povratnih klicev (angl. *asynchronous callback*). Glavna cilja omenjene knjižnice sta

doseganje visoke stopnjevanosti obdelovanja zahtev in poenostavljen programski vmesnik, ki ga prinaša večnitno programiranje.

Za podvajanje spletnih zahtev že obstajajo knjižnice Kage, Experella-Proxy in EM-Proxy [21, 12, 11], pri čemer sta prvi dve nadgradnji zadnje. Obe knjižnici ponujata razčlenjevalnik (angl. *parser*) spletnih zahtev, ki razvijalcem omogoča lažje upravljanje in spreminjanje polj v glavah spletnih zahtev. Poleg tega ponujata tudi druge napredne mehanizme, ki pa jih za implementacijo senčenja nismo zares potrebovali, zaradi česar smo se odločili za uporabo knjižnice EM-Proxy. Slednja je najbolj aktivno posodabljana, poleg tega pa za naše potrebe ponuja zadosten nabor funkcionalnosti.

3.4.1 Knjižnica EM-Proxy

Knjižnica EM-Proxy nudi domensko specifični jezik (angl. *domain specific language*) za implementacijo *posredniških strežnikov za senčenje* (angl. *shadow proxy server*) zahtev HTTP, osnovanih na knjižnici EventMachine. Z njeno pomočjo smo v magistrskem delu implementirali naš posredniški strežnik za senčenje spletnih zahtev.

EM-Proxy za odzivanje na dogodke, povezanimi z zahtevami, ponuja tri povratne klice:

- **on_data**, ki se sproži ob prejetju spletne zahteve,
- **on_response**, ki se sproži ob prejetju spletnega odgovora posameznega aplikacijskega strežnika,
- **on_finish**, ki se sproži ob zaključku pošiljanja spletnega odgovora odjemalcu.

Implementacija logike, ki se izvede znotraj zgoraj navedenih povratnih klicev, predstavlja bistvo našega posredniškega strežnika za senčenje spletnih zahtev iz produkcijskega okolja. Dejanski logiki v povratnih klicih se bomo posvetili v razdelku 3.4.4, po tem, ko bomo razložili različna načina delovanja posredniškega strežnika za senčenje.

Senčeni posredniški strežniki, implementirani s pomočjo knjižnice EM-Proxy, lahko podvajajo spletne zahteve na poljubno število aplikacijskih strežnikov. V nadaljevanju podajamo implementacijo preprostega posredniškega strežnika, ki spletne zahteve pošilja dvema aplikacijskima strežnikoma (senčenemu in produkcijskemu), uporabniku pa vrne samo odgovor produkcijskega.

```
Proxy.start(:host => "0.0.0.0", :port => 80) do |conn|
  conn.server :shadow, :host => 'localhost', :port => 3000
  conn.server :production, :host => 'localhost', :port => 3001

  conn.on_data do |data|
```

```
    data
  end
  conn.on_response do |backend, resp|
    resp if backend == :production
  end
  conn.on_finish do |backend, name|
    unbind if backend == :production
  end
end.
```

3.4.2 Nadgradnja posredovanja zahtev

Knjižnica EM-Proxy podpira dva različna načina podvajanja zahtev na aplikacijske strežnike. V primeru, da v povratnem klicu `on_data` vrnemo le spletno zahtevo, se enako zahtevo pošlje vsem aplikacijskim strežnikom, ki smo jih registrirali s posredniškim strežnikom. Če pa v povratnem klicu `on_data` poleg vsebine spletne zahteve podamo tudi seznam izbranih aplikacijskih strežnikov, se zahtevo posreduje le aplikacijskim strežnikom iz omenjenega seznama. Ne glede na to, ali zahtevo pošljemo vsem aplikacijskim strežnikom ali pa zgolj podmnožici, se vsem strežnikom posredujejo spletne zahteve, ki so popolne kopije.

Zaradi potrebe po posredovanju podobnih, a ne enakih spletnih zahtev različnim aplikacijskim strežnikom, smo knjižnico EM-Proxy nadgradili. Po implementaciji nadgradnje je mogoče v povratnem klicu `on_data` za vsakega od aplikacijskih strežnikov, registriranih z našim posredniškim strežnikom za senčenje, podati lastno spletno zahtevo, ki naj se posreduje danemu aplikacijskemu strežniku. Opisana nadgradnja nam je v nadaljevanju omogočila spreminjanje glav spletnih zahtev pred posredovanjem senčenemu aplikacijskemu strežniku, ki je bilo nujno za pravilno zaznavanje regresij v ciljni spletni aplikaciji.

3.4.3 Način obdelave spletnih zahtev

Posredniški strežnik za senčenje podpira dva načina delovanja, glede to, ali senčimo samo zahteve ki so *varne* ali tudi *nevarne*; *pasivnega* in *aktivnega*. Varnost spletnih zahtev in oba načina delovanja posredniškega strežnika za senčenje podrobneje predstavimo v nadaljevanju.

Varnost spletnih zahtev

Spletna zahteva je glede na definicijo protokola HTTP verzije 1.1 *varna*, če odjemalec ne zahteva in ne pričakuje spremembe stanja aplikacijskega strežnika kot posledico obdelave

zahteve [20]. Konkretno gre za spletne zahteve z metodami protokola HTTP GET, HEAD, OPTIONS in TRACE.

Nasprotno pa obdelava *nevarnih* zahtev lahko spremeni stanje aplikacijskega strežnika. V tem primeru spletna zahteva uporablja metode protokola HTTP POST, PUT, PATCH ali DELETE. Spreminjanje stanja aplikacijskega strežnika v kontekstu spletnih aplikacij običajno obsega spreminjanje podatkov v pripadajoči podatkovni bazi.

Pasivni način

Pasivni način delovanja se uporablja za senčenje *varnih* spletnih zahtev. V tem načinu delovanja posredniški strežnik za senčenje senčenemu strežniku posreduje zgolj varne, produkcijskemu strežniku pa tako varne kot tudi nevarne spletne zahteve. Za zagotavljanje pravilnega delovanja spletne aplikacije mora namreč produkcijski aplikacijski strežnik obdelovati vse spletne zahteve, ki mu jih dodeli spletni strežnik.

Replikacija podatkovne baze v pasivnem načinu Posledica obdelave nevarnih spletnih zahtev s strani enega ali več produkcijskih aplikacijskih strežnikov so spremembe produkcijske podatkovne baze, do katerih v splošnem prihaja tudi med postopkom senčenja. Ker lahko spremembe produkcijske podatkovne baze vplivajo na rezultat obdelave varnih zahtev, moramo poskrbeti za replikacijo nadrejene podatkovne baze iz produkcijskega okolja v podrejeno podatkovno bazo v senčenem okolju. Replikacija se izvede ob vsaki zaznani spremembi produkcijske podatkovne baze, zato aplikacijski strežniki iz obeh okolij dostopajo do instanc podatkovne baze, ki sta popolni kopiji. Posledično obe podatkovni bazi vračata enake rezultate poizvedb, odgovora obeh aplikacijskih strežnikov pa bi morala biti enaka. Slednje nam omogoča enostavno preverjanje pravilnosti produkcijske in novejše (tj. testirane) izdaje, saj razlike v odgovorih obeh aplikacijskih strežnikov zagotovo niso posledica razlik v podatkih obeh podatkovnih baz, to pa lahko pomeni regresijo. Kot bomo videli v razdelku 3.6.2, lahko v splošnem še vedno pride do izjeme, ko replicirana podatkovna baza ni popolna kopija produkcijske, kar vodi v napačno zaznane regresije.

Aktivni način

Aktivni način delovanja je namenjen senčenju tako varnih kot tudi *nevarnih* spletnih zahtev. Ta način na račun večje kompleksnosti omogoča bolj celovito testiranje ciljne spletne aplikacije. Omogoča namreč senčenje tudi tistih spletnih zahtev, ki tekom obdelave v aplikacijskem strežniku spreminjajo stanje podatkovne baze z vnašanjem, spreminjanjem ali brisanjem zapisov.

Kljub temu, da je HTTP protokol brez pomnenja (angl. *stateless*), aplikacijski strežnik za pravilno obdelavo spletne zahteve pogosto potrebuje podatke, ki izhajajo iz predhodne komunikacije z odjemalcem. To aplikacijskemu strežniku omogoča, da na nek način drži sejo z uporabnikom, kar navadno implementiramo s pomočjo *piškotkov* (angl. *cookies*). Piškotek ustvari aplikacijski strežnik in ga v spletnem odgovoru posreduje odjemalcu. Spletni odjemalec piškotek shrani in ga doda v glavo vsake nadaljnje zahteve na isti strežnik. Spletna aplikacija lahko piškotke med drugim uporablja za hranjenje žetonov sej, ki so namenjeni avtentikaciji zahtev s strani aplikacijskega strežnika.

Piškotka, ki ju aplikacijska strežnika iz različnih okolij hranita v podatkovni bazi za istega uporabnika, sta seveda različna, vendar pa do končnega uporabnika pride samo odgovor aplikacijskega strežnika iz produkcijskega okolja. Za nadaljnjo komunikacijo z aplikacijo se zato seveda uporabi piškotek, ki ga je izdal produkcijski aplikacijski strežnik, kar pa ne zadostuje za pravilno vzpostavitev uporabniške seje z aplikacijskim strežnikom v senčenem okolju. V aktivnem načinu mora posredniški strežnik za senčenje za vzpostavitev uporabniške seje s senčenim aplikacijskim strežnikom:

1.) hraniti žetone za držanje uporabniških sej, ki jih je izdal senčeni aplikacijski strežnik,

2.) pred pošiljanjem senčenemu strežniku v podvojenih zahtevah zamenjati žetone v piškotkih, ki jih je izdal produkcijski aplikacijski strežnik, z zadnjimi, ki jih je izdal aplikacijski strežnik v senčenem okolju.

Posredniški strežnik za senčenje mora torej spreminjati glave zahtev, zaradi česar opisani način delovanja imenujemo aktivni način.

Replikacija podatkovne baze v aktivnem načinu Rezultat obdelave nevarne spletne zahteve s strani aplikacijskega strežnika je praviloma sprememba v podatkovni bazi. Preden posredniški strežnik za senčenje posreduje nevarno spletno zahtevo aplikacijskima strežnikoma, najprej prekine nadaljnjo replikacijo produkcijske podatkovne baze v senčeno okolje. S tem zagotovimo, da obdelava nevarne zahteve s strani produkcijskega strežnika ne povozi sprememb v senčeni podatkovni bazi, saj se replikacija ob spremembi produkcijske podatkovne baze ne bo izvedla. Produkcijski in senčeni strežnik bosta tako pred začetkom obdelave dane nevarne zahteve imela enaki podatkovni bazi, med obdelavo nevarne zahteve pa bo spreminjal vsak svojo.

Prehajanje med pasivnim in aktivnim načinom

Posredniški strežnik za senčenje mora v aktivnem načinu delovanja praviloma ob vsaki nevarni zahtevi ponastaviti senčeno podatkovno bazo in sprožiti začetno replikacijo produkcijske podatkovne baze v senčeno, da sta vsebini obeh podatkovnih baz pred obdelavo zahteve enaki. Posledično je senčenje v aktivnem načinu delovanja posredniškega strežnika za

senčenje počasnejše kot v pasivnem načinu delovanja, saj zahteva ponastavitev in začetno replikacijo, ki skupno trajata reda deset sekund (za razliko od nekaj milisekund, sicer potrebnih za senčenje poljubne zahteve brez ponastavitve in začetne replikacije). Število nevarnih spletnih zahtev, ki jih posredniški strežnik lahko senči v danem časovnem okviru, je torej v veliki meri odvisno od časa, potrebnega za ponastavitev in začetno replikacijo podatkovne baze v senčenem okolju. Po drugi strani pa režijsko delo zaradi ponastavitve in replikacije podatkovne baze, ki ga zahteva senčenje nevarnih spletnih zahtev, ni potrebno tudi za varne zahteve.

V ta namen smo za zagotavljanje večjega pretoka senčenih zahtev implementirali dinamično prehajanje med pasivnim in aktivnim načinom delovanja posredniškega strežnika za senčenje. Posredniški strežnik za senčenje tako nikoli ne deluje zgolj v pasivnem ali zgolj v aktivnem načinu, pač pa ima vgrajeno preprosto logiko za prehajanje med obema načinoma.

Posredniški strežnik privzeto deluje v pasivnem načinu, ki je sicer bolj učinkovit, vendar lahko senči samo varne spletne zahteve. Ko pa iz odgovora na varno zahtevo sklepa, da bo naslednja zahteva nevarna (npr. na podlagi žetona CSRF v vsebini HTML), preide v aktivni način delovanja. Ob prehodu v aktivni način delovanja posredniški strežnik prekine nadaljnjo replikacijo senčene podatkovne baze in nastavi interni *števec nevarnih spletnih zahtev* na vrednost 1. Omenjeni števec posredniški strežnik v nadaljevanju poveča za 1 ob vsaki naslednji nevarni zahtevi. Dokler števec ne doseže mejne vrednosti N (gre za nastavljivo vrednost, npr. $N = 5$), se za nadaljnje nevarne zahteve ne bo izvedla ponovna ponastavitev in začetna replikacija, ampak se bo uporabila kar podatkovna baza, ki je bila vzpostavljena ob prehodu v aktivni način, vključujoč spremembe, ki so nastale kot rezultat obdelave prejšnjih nevarnih zahtev. Ko števec nevarnih zahtev doseže mejno vrednost N, posredniški strežnik izvede ponastavitev senčene podatkovne baze in začetno replikacijo produkcijske baze v senčeno okolje. Nato se avtomatsko vrne v pasivni način delovanja, v katerem se ponovno izvaja replikacija produkcijske baze v senčeno okolje ob vsaki zaznani spremembi, vrednost števca nevarnih zahtev pa se postavi na 0.

Vnos napak Zavedati se moramo dejstva, da je mejna vrednost števca nevarnih zahtev, N, kompromis med učinkovitostjo delovanja posredniškega strežnika za senčenje in kasnejšo pravilnostjo zaznavanja regresij. Vsebina produkcijske in senčene podatkovne baze se tekom obdelave N nevarnih zahtev neodvisno spreminja. V primeru, da pri obdelavi nevarne spletne zahteve v senčenem okolju pride do regresije, se v podatkovno bazo senčenega okolja lahko zapiše drugačna (napačna) vrednost kot v produkcijsko. Regresija v odgovoru dane spletne zahteve zato lahko vpliva na obdelavo vseh nadaljnjih zahtev, ki se izvedejo pred prehodom posredniškega strežnika v pasivni način delovanja. Posledično se lahko v postopku analize poleg odgovora na i-to zahtevo kot potencialne regresije napačno

klasificirajo tudi odgovori na nadaljnjih $N - i$ zahtev, četudi regresij ne vsebujejo.

3.4.4 Logika v povratnih klicih

Knjižnica EM-Proxy implementira povratne klice, ki se prožijo v različnih stopnjah življenjskega cikla spletne zahteve. Z njihovo uporabo smo implementirali programsko logiko našega posredniškega strežnika za senčenje.

Povratni klic on_data

V primeru, da se začetna replikacija produkcijske podatkovne baze v senčeno še ni zaključila, posredniški strežnik za senčenje zahtev ne glede na način delovanja (pasivni ali aktivni) senčenja še ne more pravilno izvajati, zato prejeto spletno zahtevo zgolj posreduje v produkcijsko okolje.

V nasprotnem primeru pa posredniški strežnik za senčenje iz zahteve prebere spletno pot, tip spletne zahteve in identifikator zahteve, opisan v razdelku 3.2.

V primeru, da je zahteva nevarna, se poveča števec nevarnih zahtev. Posredniški strežnik za senčenje nato pregleda, ali spletna zahteva vsebuje katerega od prednastavljenih tipov žetonov za držanje sej, ki jih je predhodno shranil iz odgovorov senčenega okolja. Če je tip žetona prisoten in v podatkovni zbirki Redis obstaja zapis za spletno zahtevo z danim omrežnim naslovom in tipom žetona, se vsebina žetona zamenja. Če lahko tip žetona v sklopu zahteve HTTP vsebuje neveljavne znake, se žeton še kodira s kodiranjem `CGI`. Opisan postopek zamenjave žetonov v aktivnem načinu delovanja posredniškega strežnika implementira spodnja programska koda:

```
REPLACE_REGEX = {'remember_token'=>/remember_token=(.*?)(;|\r)/}
SCAN_REGEX = {'remember_token'=>/remember_token=(.*?); path/}
ESCAPED_TOKENS = []
redis = Redis.new(:host=>"127.0.0.1", :port=>6379, :db => 0)
ip = request_id.split('-')[4..7].join('.')
SCAN_REGEX.keys.each do |token_name|
    token = data.scan(REPLACE_REGEX[token_name])
    if token.size > 0
      stored_token = redis.hget(ip, token_name)
      if stored_token
        escaped_token = stored_token
        if ESCAPED_TOKENS.include?(token_name)
          escaped_token = CGI.escape(stored_token)
        end
```

```
        data = data.gsub(token[0][0], escaped_token)
      end
    end
end
data.
```

V primeru uporabe aktivnega načina se v produkcijsko okolje nato pošlje originalna zahteva, v senčeno okolje pa spremenjena zahteva. Če pa posredniški strežnik za senčenje deluje v pasivnem načinu, se v obe okolji pošlje originalna zahteva.

Povratni klic on_response

Ne glede na to, ali posredniški strežnik za senčenje zahtev prejme odgovor produkcijskega ali senčenega aplikacijskega strežnika, prejeti spletni odgovor zapiše v medpomnilnik. V primeru, da je prejeti odgovor prišel iz produkcijskega okolja, ga posredniški strežnik posreduje spletnemu strežniku, ta pa končnemu uporabniku.

Povratni klic on_finish

Po tem, ko posredniški strežnik za senčenje spletni odgovor uspešno posreduje spletnemu strežniku, prekine pripadajočo povezavo protokola HTTP. Nato v podatkovno zbirko Redis shrani spletna odgovora tako produkcijskega kot tudi senčenega strežnika, kjer počakata na kasnejšo analizo. Posredniški strežnik za senčenje nato preveri, ali spletni odgovor senčenega strežnika vsebuje katerega od prednastavljenih tipov žetonov, namenjenega držanju uporabniške seje. V primeru, da spletni odgovor senčenega strežnika vsebuje tak žeton, se slednji za potrebe morebitne ponovne vzpostavitve seje s senčenim strežnikom shrani v podatkovno zbirko Redis, pri čemer se kot ključ uporabi omrežni naslov izvorne spletne zahteve.

Če posredniški strežnik deluje v pasivnem načinu, sledi še preverjanje vsebine odgovorov za elemente HTML, na podlagi katerih lahko sklepamo, da bo naslednja zahteva nevarna. Če so v odgovoru prisotni na primer žetoni CSRF, posredniški strežnik ustavi replikacijo podrejene podatkovne baze v senčenem okolju in sproži prehod v aktivni način.

V zadnjem koraku povratnega klica se v primeru delovanja posredniškega strežnika v aktivnem načinu preveri še število obdelanih nevarnih spletnih zahtev. V primeru, da je vrednost števca nevarnih zahtev večja od nastavljene mejne vrednosti, se v sklopu prehoda iz aktivnega v pasivni način sprožita ponastavitev senčene podatkovne baze in začetna replikacija produkcijske baze v podatkovno bazo senčenega okolja.

3.5 Beleženje aplikacijskih metrik

Da lahko naša rešitev avtomatsko zaznava zmogljivostne regresije v ciljni spletni aplikaciji, od aplikacije potrebuje določene podatke. Ti podatki so t. i. *aplikacijske metrike*, ki jih beleži aplikacijski strežnik. Zanimajo nas predvsem naslednje zmogljivostne metrike, ki opisujejo obremenitev aplikacijskih in podatkovnih strežnikov:

- število poizvedb po podatkovni bazi,
- skupni čas izvajanja poizvedb po podatkovni bazi,
- čas, potreben za izris pogleda (vsebine, ki jo končni uporabnik vidi v spletnem brskalniku),
- skupni čas, potreben za obdelavo zahteve.

3.5.1 Implementacija

Za spremljanje in analizo še veliko večjega nabora aplikacijskih metrik kot je zgornji, pa tudi sistemskih metrik, že obstajajo celovite rešitve kot sta NewRelic [28] in Skylight [36]. Ker pa smo potrebovali le zelo majhen del njunih funkcionalnosti, smo se odločili za razvoj lastne, lažje (angl. *lightweight*) programske knjižnice. Poleg tega je uporaba lastne knjižnice poenostavila kasnejšo integracijo naše rešitve s ciljno spletno aplikacijo.

Pri implementaciji beleženja aplikacijskih metrik smo si pomagali s programskim vmesnikom `ActiveSupport::Notifications`, ki ga ponuja ogrodje Ruby on Rails. Slednji omogoča obveščanje o internih dogodkih ogrodja, tj. o dogodkih, ki izvirajo iz ogrodja Ruby on Rails tekom obdelave spletnih zahtev. Pregledali smo ponujen nabor omenjenih dogodkov in izmed njih izbrali take, ki se nanašajo na poizvedbe po podatkovni bazi in na izvrševanje aplikacijske logike.

Beleženje aplikacijskih metrik smo nato implementirali na nivoju vmesne programske opreme (angl. *middleware*) med aplikacijskim strežnikom in aplikacijsko logiko. Tako vmesno programje omogoča spreminjanje spletnih zahtev ali odgovorov (npr. dodajanje polj glave), pred oziroma po tem, ko jih uporabi aplikacijska logika. Pri tem smo sledili enostavnemu programskemu vmesniku Rack [14], napisanem v programskem jeziku Ruby. Vmesnik Rack za implementacijo vmesne programske kode predpisuje uporabo metode `call`, ki mora:

- kot argument sprejeti podatke o spletni zahtevi,
- kot rezultat vrniti podatke o spletnem odgovoru, ki ga oblikuje aplikacija, in sicer v obliki polja s tremi elementi; statusom spletnega odgovora, polji glave spletnega odgovora in vsebino spletnega odgovora.

Našo rešitev za sporočanje aplikacijskih metrik smo osnovali na obstoječi odprtokodni implementaciji vmesne programske opreme, `server_timings_middleware` [34]. Slednjo smo nadgradili tako, da omogoča izbiro dogodkov ogrodja Ruby on Rails, za katere želimo prejemati obvestila. Ogrodje Ruby on Rails med obdelavo spletne zahteve s strani aplikacije vmesno programje obvešča o izbranih dogodkih, po obdelavi spletne zahteve pa jih vmesno programje po potrebi agregira in vstavi v spletni odgovor v treh dodatnih poljih glave:

- `X-Sql-Queries`, ustreza aplikacijski metriki *število poizvedb po podatkovni bazi*,
- `X-Db-Runtime`, ustreza aplikacijski metriki *skupni čas izvajanja poizvedb po podatkovni bazi*,
- `X-View-Runtime`, ustreza aplikacijski metriki *čas, potreben za izris pogleda.*

Na tem mestu omenimo še, da vrednost še zadnje izbrane aplikacijske metrike (*skupni čas, potreben za obdelavo zahteve*) v spletni odgovor samodejno vstavi ogrodje Ruby on Rails v polju glave `X-Runtime`.

Programsko kodo implementirane metode `call` podajamo spodaj:

```
def call(env)
    events = []
    capture_events = [  'process_action.action_controller',
                        'sql.active_record']
    event_filter = Regexp.new(capture_events.join('|'))

    subs = ActiveSupport::Notifications.subscribe(event_filter)
    do |*args|
        events << ActiveSupport::Notifications::Event.new(*args)
    end

    status, headers, body = @app.call(env)

    ActiveSupport::Notifications.unsubscribe(subs)
    headers = set_metric_headers(headers, events)
    [status, headers, body]
end.
```

3.6 Spletna aplikacija za analizo in nadzor senčenja

Za avtomatsko zaznavanje regresij v izbrani spletni aplikaciji potrebujemo še programsko podprto primerjavo in analizo spletnih odgovorov iz senčenega in produkcijskega okolja. V ta namen smo z ogrodjem Ruby on Rails razvili spletno aplikacijo, ki poleg avtomatske analize tako vsebine spletnih odgovorov kot tudi aplikacijskih metrik iz polj glave spletnih odgovorov razvijalcem ponuja tudi preprosto nadzorno ploščo (angl. *dashboard*). Spletna aplikacija za analizo in nadzor senčenja se izvaja na istem strežniku kot posredniški strežnik za senčenje.

3.6.1 Uvoz vhodnih podatkov

Kot rečeno v razdelku 3.4.4, posredniški strežnik za senčenje obe različici spletnih zahtev in spletnih odgovorov, tj. zahtevo in odgovor produkcijskega ter zahtevo in odgovor senčenega strežnika, zaradi potrebe po čim hitrejši obdelavi shrani v podatkovno zbirko Redis. Spletna aplikacija za analizo in nadzor senčenja podatke periodično uvaža iz podatkovne zbirke Redis v lastno podatkovno bazo sistema za upravljanje podatkovnih baz PostgreSQL. S tem namreč pridobimo možnost enostavne obdelave podatkov v spletni aplikaciji zaradi preslikovanja podatkov v objekte programskega jezika (angl. *object-relational mapping*). Dodaten razlog za uvoz podatkov iz zbirke tipa ključ-vrednost je ta, da za samo analizo ni več potrebe po čim hitrejšem vnašanju zapisov, saj se slednja ne izvaja tekom obdelave produkcijske spletne zahteve in zato ne vpliva na končne uporabnike.

Posamezen zapis v podatkovni zbirki Redis vsebuje naslednje podatke:

- identifikator spletne zahteve,
- čas, ko je bila spletna zahteva poslana,
- pot iz spletne zahteve,
- tip spletne zahteve (metoda protokola HTTP),
- zahtevo, poslano v produkcijsko okolje,
- zahtevo, poslano v senčeno okolje,
- odgovor iz produkcijskega okolja,
- odgovor iz senčenega okolja,
- identifikator izdaje (zgoščena vrednost uveljavitve spremembe).

Iz polj glave spletnih odgovorov obeh aplikacijskih strežnikov spletna aplikacija za analizo nato prebere še aplikacijske metrike, dodane s strani vmesnega programja za sporočanje aplikacijskih metrik.

3.6.2 Šum v spletnih odgovorih

Z analizo spletnih odgovorov senčenega in produkcijskega strežnika želimo poiskati razlike v odgovorih, ki se pojavijo zaradi spremembe izvorne kode spletne aplikacije ali njene konfiguracije. Do razlik pa prihaja tudi med odgovori različnih produkcijskih aplikacijskih strežnikov na enako spletno zahtevo, torej kljub identični izvorni kodi in konfiguraciji. Razlike, ki so v spletnih odgovorih vedno prisotne, ne glede na to ali se pojavljajo v poljih glave ali v vsebini, imenujemo *šum v spletnih odgovorih*.

Obravnava šumnih podatkov v spletnih odgovorih senčenega in produkcijskega strežnika in njihovo izločanje iz analiz je bistvenega pomena za pravilno zaznavanje potencialnih regresij. Pri izločanju šuma smo si pomagali s *pravili*, ki smo jih v aplikaciji za analizo spletnih odgovorov sicer v prvi vrsti implementirali za klasifikacijo potencialnih regresij, ki so posledica prehoda ciljne aplikacije na novo verzijo. Pravila podrobneje predstavimo v razdelku 3.6.3.

Ob odsotnosti šuma bi bilo zaznavanje regresij trivialno, saj bi vsaka razlika v spletnih odgovorih produkcijskega in senčenega aplikacijskega strežnika na enako spletno zahtevo pomenila regresijo.

V nadaljevanju predstavimo nekaj kategorij šumnih podatkov v spletnih odgovorih.

Časovni žigi

Uporaba protokola NTP v povprečju uvede nekaj deset milisekund razlike v vrednosti časovnih žigov spletnih odgovorov strežnikov v razpršenih geografskih legah, oziroma nekaj milisekund razlike v primeru, da so strežniki v lokalnem omrežju [25]. Čeprav se je razlikam v časovnih žigih v praksi mogoče izogniti, je slednje nepraktično, saj zahteva drago namensko strojno opremo.

Uporaba psevdonaključnih vrednosti

Tako vsebina kot tudi nekatera polja glave spletnih odgovorov lahko temeljijo na vrednostih, ki jih vračajo generatorji psevdonaključnih števil. Ujemajoče se vrednosti v spletnih odgovorih različnih aplikacijskih strežnikov sicer načeloma lahko dosežemo z uporabo istega semena (angl. *seed*) za generator psevdonaključnih števil, vendar le v primeru, da je število klicev psevdonaključnega generatorja s strani obeh aplikacij ves čas enako. V praksi je slednje skoraj nemogoče doseči, saj produkcijski strežnik deluje povsem neodvisno

od senčenega.

Pri usklajevanju psevdonaključnih vrednosti v spletnih odgovorih različnih strežnikov pa še večjo težavo predstavlja uporaba generatorjev psevdonaključnih števil v zunanjih knjižnicah, tj. knjižnicah, ki jih uporablja dana aplikacija. Tudi če bi ob vsaki novi verziji spletne aplikacije želeli popravljati semena generatorjev psevdonaključnih števil pri klicih relevantnih zunanjih knjižnic, ni rečeno, da zunanje knjižnice to sploh omogočajo.

Klici zunanjih storitev

Obdelava spletne zahteve lahko obsega klic zunanje storitve, čigar rezultat lahko vključimo v spletni odgovor ciljne spletne aplikacije. Odgovor aplikacijskega strežnika je v tem primeru odvisen od rezultata klica zunanje storitve. Dva klica iste zunanje storitve v časovnem razmaku zgolj nekaj milisekund lahko vrneta različen rezultat. Prav tako lahko do razlik v spletnih odgovorih aplikacijskih strežnikov pride, ker zunanja storitev določenemu aplikacijskemu strežniku v danem trenutku ni nujno dosegljiva.

Neusklajenost zaradi replikacije podatkovne baze

Replikacija sprememb iz produkcijske podatkovne baze v senčeno okolje nujno zahteva nekaj časa. Od trenutka, ko se sprememba danega podatka uveljavi v produkcijski podatkovni bazi, do trenutka, ko je spremenjena vrednost na voljo v senčeni podatkovni bazi, lahko nova nevarna spletna zahteva spremeni isti podatek v produkcijski bazi. Zaradi slednjega lahko v pasivnem načinu delovanja posredniškega strežnika za senčenje produkcijski in senčeni aplikacijski strežnik vrneta različna odgovora, saj produkcijski strežnik uporabi novo vrednost podatka, senčeni pa prejšnjo.

3.6.3 Zaznavanje vsebinskih regresij

Za zaznavanje vsebinskih regresij spletne aplikacije moramo pregledati tako vsebino kot tudi polja glav spletnih odgovorov produkcijskega in senčenega strežnika. V naših analizah smo razlike v spletnih odgovorih zaznavali na nivoju posameznih vrstic, pri čemer smo si pomagali s knjižnico Diffy [15]. Predstavljajmo si spletni odgovor kot besedilo; v tem primeru knjižnica Diffy omogoča odkrivanje vrstic v odgovoru senčenega strežnika, ki so bile dodane, odstranjene ali spremenjene glede na odgovor produkcijskega strežnika. Kot rečeno v prejšnjem razdelku pa zgolj osnovno prepoznavanje razlik v spletnih odgovorih za zaznavanje regresij ne zadošča, saj razlike zaradi šuma napačno klasificira kot regresije.

Pravila

Pravilno zaznavanje vsebinskih regresij v ciljni spletni aplikaciji zahteva natančnejšo klasifikacijo sprememb v odgovorih produkcijskega in senčenega strežnika, zaradi česar smo na nivoju aplikacije za analizo in nadzor senčenja implementirali t. i. *pravila*.

Pravila imajo naslednje atribute:

- ime,
- regularni izraz, s katerim določimo, na kateri del spletnega odgovora se pravilo nanaša,
- pričakovano spremembo vsebine, določene z regularnim izrazom. Lahko ima vrednosti `modify` (poljubna sprememba), `add` (dodajanje) ali `remove` (odstranitev), pri čemer za referenco vzamemo odgovor produkcijskega strežnika,
- oceno spremembe, izraženo z negativnim številom (regresija) ali nič (ignoriramo). Pravila z večjimi absolutnimi vrednostmi ocen imajo večji doprinos k oceni izdaje.

Na primer, če želimo tekom preverjanja pravilnosti nove izdaje aplikacije izločiti vse spremembe elementa spletne strani (slednja se nahaja v vsebini spletnega odgovora), ki hrani žeton CSRF, dodamo pravilo z regularnim izrazom

```
<meta content=(.)* name="csrf-token" />,
```

pričakovano spremembo `modify` in oceno spremembe 0.

Spremembe med odgovoroma senčenega in produkcijskega okolja, ki jih obstoječa pravila ne zajemajo, smatramo kot regresije. Vsaka nepredvidena dodana, odstranjena ali spremenjena vrstica odgovora zato avtomatsko dobi svoje pravilo z oceno spremembe -1. Ob koncu analize spletnih odgovorov produkcijskega in senčenega strežnika seštejemo ocene pravil vseh zaznanih sprememb. Odgovori z negativnim seštevkom ocen sprememb se označijo kot regresije.

Privzeta pravila

Ob uvozu podatkov o spletnih zahtevah in odgovorih produkcijskega in senčenega strežnika v aplikacijo za analizo se avtomatsko ustvari začetni nabor privzetih pravil. Vsa ta pravila imajo oceno spremembe 0 (kar pomeni, da pripadajoče spremembe ignoriramo), saj so namenjena izločevanju šuma v odgovorih iz analiz. Privzeta pravila se nanašajo tako na polja glave spletnih odgovorov kot tudi na njihovo vsebino, in obsegajo naslednje podatke:

- polje glave `ETag`, ki se uporablja kot identifikator verzije spletnega vira za namene predpomnenja,

- polje glave `SetCookie`, ki vsebuje uporabniške piškotke,
- značka HTML `meta` z žetonom CSRF, ki je namenjen preprečevanju napadov ponarejanja spletnih strani,
- žeton CSRF, namenjen preprečevanju napadov ponarejanja spletnih strani, ki se nahaja v znački HTML vnosnega obrazca,
- polja glave z aplikacijskimi metrikami `X-Runtime`, `X-View-Runtime`, `X-Db-Runtime` in `X-Sql-Queries`.

Z izjemo polj glave z aplikacijskimi metrikami, ki so posebnost naše rešitve, smo za privzeta pravila uporabili prav zgornje podatke, saj se ti v splošnem pogosto pojavljajo v večini spletnih aplikacij. Polja glave z aplikacijskimi metrikami iz analize za zaznavanje vsebinskih regresij izdaje izločimo, saj jih obravnavamo ločeno, v okviru zaznavanja zmogljivostnih regresij izdaje.

Dodajanje lastnih pravil

Poleg privzetih pravil se za oceno posameznih odgovorov uporabljajo tudi pravila, ki jih lahko naknadno vnese razvijalec. Slednjih se ne da posplošiti za več ciljnih spletnih aplikacij - odvisne so namreč od aplikacijske logike ciljne spletne aplikacije, ki pa jo najbolje pozna razvijalec. Z uporabo nadzorne plošče, ki jo predstavimo v razdelku 3.6.5, lahko razvijalec dodaja tri tipe pravil:

- pravila za izločanje šuma (z oceno spremembe 0),
- pravila za izločanje pričakovanih sprememb nove izdaje (z oceno spremembe 0),
- pravila za zaznavanje regresij ključnih delov aplikacije (z zelo majhno negativno oceno spremembe).

3.6.4 Zaznavanje zmogljivostnih regresij

Kot rečeno, si pri zaznavanju zmogljivostnih regresij pomagamo z aplikacijskimi metrikami, ki se nahajajo v glavah spletnih odgovorov obeh aplikacijskih strežnikov. Za analizo je ključnega pomena, da zabeleženih vrednosti posameznih aplikacijskih metrik ne obravnavamo preveč strogo. Vrednosti določenih aplikacijskih metrik se namreč ne bodo nujno ujemale niti, če jih primerjamo med dvema produkcijskima strežnikoma, saj so odvisne od dejavnikov, kot sta stanje omrežja in obremenitev strežnika. Zaradi slednjega pri odkrivanju zmogljivostnih regresij postopamo nekoliko drugače kot pri odkrivanju vsebinskih regresij, saj za vsako aplikacijsko metriko definiramo največje dovoljeno odstopanje vrednosti v senčenem okolju glede na produkcijsko okolje.

Privzete vrednosti dovoljenih odstopanj aplikacijskih metrik so naslednje:

- največ 100 milisekund za skupni čas obdelave spletne zahteve (polje glave `X-Runtime`),
- največ 10 dodatnih ali odstranjenih poizvedb po podatkovni bazi (polje glave `X-Db-Queries`),
- največ 50 milisekund za skupni čas izvajanja poizvedb po podatkovni bazi (polje glave `X-Db-Runtime`),
- največ 50 ms za čas, potreben za izris pogleda (polje glave `X-View-Runtime`).

Vrednosti dovoljenih odstopanj so sicer odvisne od zahtev posamezne aplikacije in jih je mogoče ustrezno nastaviti. Zavedati se moramo, da premajhna dovoljena odstopanja vrednosti aplikacijskih metrik lahko vodijo v napačno zaznane zmogljivostne regresije novih izdaj, medtem ko lahko prevelika dovoljena odstopanja potencialne zmogljivostne regresije zgrešijo.

V primeru, da vsaj ena od aplikacijskih metrik presega dovoljeno odstopanje, spletnemu odgovoru oceno zmanjšamo za 20 točk.

3.6.5 Nadzorna plošča

Potek in rezultate preverjanja pravilnosti testirane izdaje spletne aplikacije si razvijalec lahko ogleda na nadzorni plošči, ki je del spletne aplikacije za nadzor in analizo senčenja.

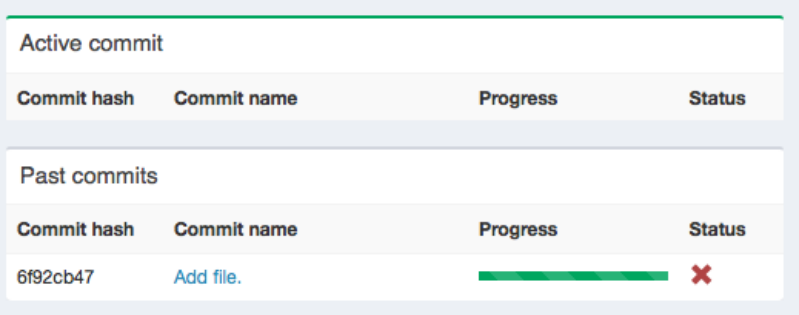
Rezultati testiranja izdaj

Na vstopni strani nadzorne plošče lahko vidimo seznam aktivnih in preteklih senčenih izdaj, kot je prikazano na sliki 3.3. Posamezno izdajo spletne aplikacije določa zgoščena vrednost uveljavitve spremembe, na seznamu pa vidimo tudi ime uveljavitve spremembe, grafični prikaz napredka testiranja in rezultat analize izdaje. Rezultat je lahko pozitiven, kar pomeni, da je izdaja pravilna, oz. da v njej nismo zaznali regresij, ali pa negativen, kar pomeni, da smo odkrili regresije.

Do podrobnosti analize posamezne izdaje pridemo s klikom na ustrezno vrstici v tabeli izdaj.

Testirane spletne poti posamezne izdaje

Na spletni strani s podrobnostmi izdaje lahko vidimo poti do spletnih virov ciljne spletne aplikacije, ki so se pojavljale v analiziranih spletnih zahtevah. Za vsako pot do spletnega vira beležimo:



Slika 3.3: Seznam testiranih izdaj ciljne spletne aplikacije in stanje napredka oz. rezultati testiranja.

- število zahtev,
- delež zahtev, za katere smo v pripadajočem odgovoru testirane izdaje (tj. v odgovoru senčenega aplikacijskega strežnika) zaznali regresijo,
- najnižjo točkovno oceno odgovora na zahtevo za dano spletno pot.

S klikom na posamezno pot do spletnega vira lahko nato pridemo na stran s podrobnostmi o spletnih zahtevah za dano pot.

Zahteve posameznih poti

Za vsako pot spletne aplikacije je na voljo seznam vseh spletnih zahtev, ki so do te poti dostopale v postopku testiranja.

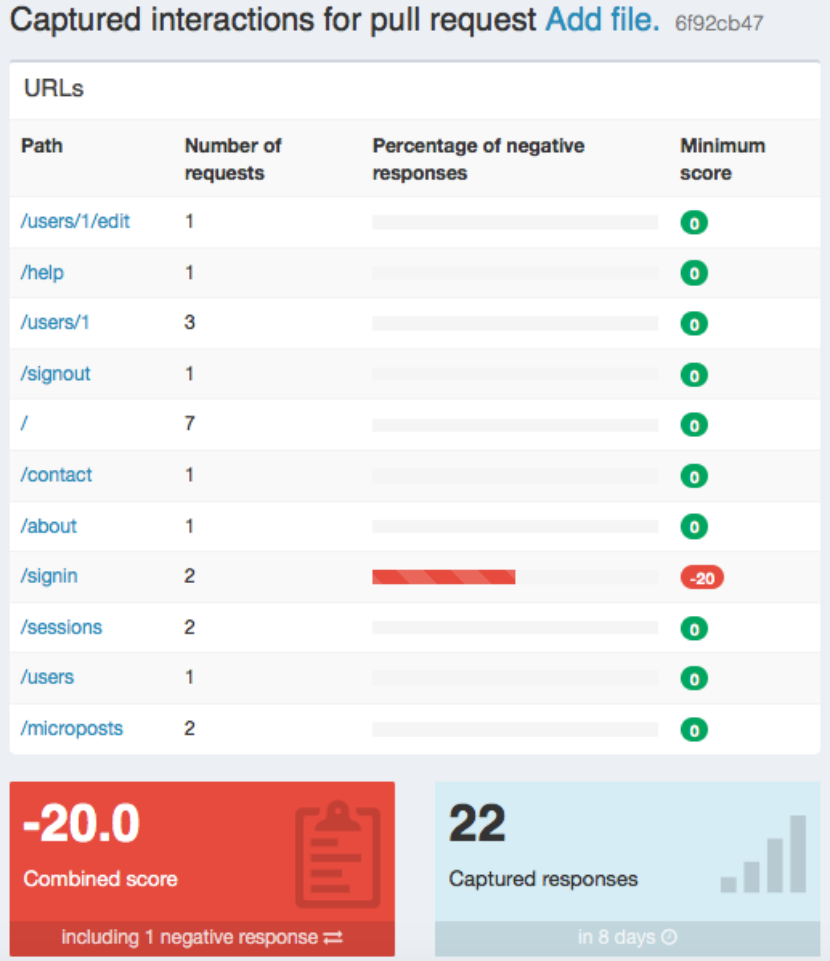
Ob vsaki zahtevi je navedena vsota točkovnih ocen sprememb na podlagi pravil za odkrivanje regresij, ki so rezultat analize spletnih odgovorov produkcijskega in senčenega strežnika na to zahtevo. Seznam spletnih zahtev, ki so v enem od testiranj dostopale do spletne poti `/signin`, prikazuje slika 3.5.

S klikom na posamezno zahtevo pridemo do podrobnih informacij o posamezni zahtevi in točkovanju sprememb v pripadajočem senčenem odgovoru.

Ocenjevanje sprememb v odgovorih na posamezno zahtevo

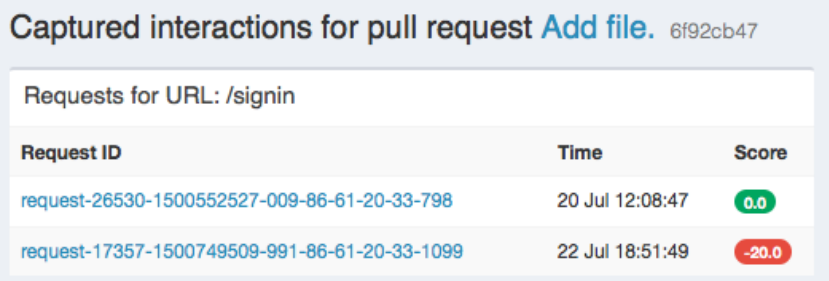
Na spletni strani s podrobnostmi analize spletnih odgovorov na posamezno zahtevo si lahko ogledamo:

- *spletni zahtevi za produkcijski in senčeni strežnik.* Razvijalec eno ob drugi vidi originalno spletno zahtevo za produkcijski strežnik in zahtevo za senčeni strežnik, ki jo je generiral posredniški strežnik za senčenje. Primer zahtev prikazuje slika 3.6.



Slika 3.4: Seznam poti do spletnih virov, do katerih so dostopale spletne zahteve v okviru testiranja posamezne izdaje.

- *Spletna odgovora produkcijskega in senčenega strežnika.* Razvijalec enega ob drugem vidi tudi spletna odgovora strežnikov, kot je prikazano na sliki 3.7. Razlike med odgovoroma so obarvane, vidimo pa lahko tako razlike na nivoju vrstic kot tudi na nivoju vsebine posameznih vrstic.
- *Vsa pravila za ocenjevanje vsebinskih razlik v spletnih odgovorih*, kot jih prikazuje slika 3.8. Gre za vsa obstoječa pravila, ki se uporabljajo za analizo specifičnega para spletnih odgovorov, in njihove atribute. Poleg tega lahko razvijalec preko nadzorne plošče na tem mestu doda novo ali izbriše obstoječe pravilo.
- *Uporabljena pravila za ocenjevanje vsebinskih razlik v spletnih odgovorih*, kot jih



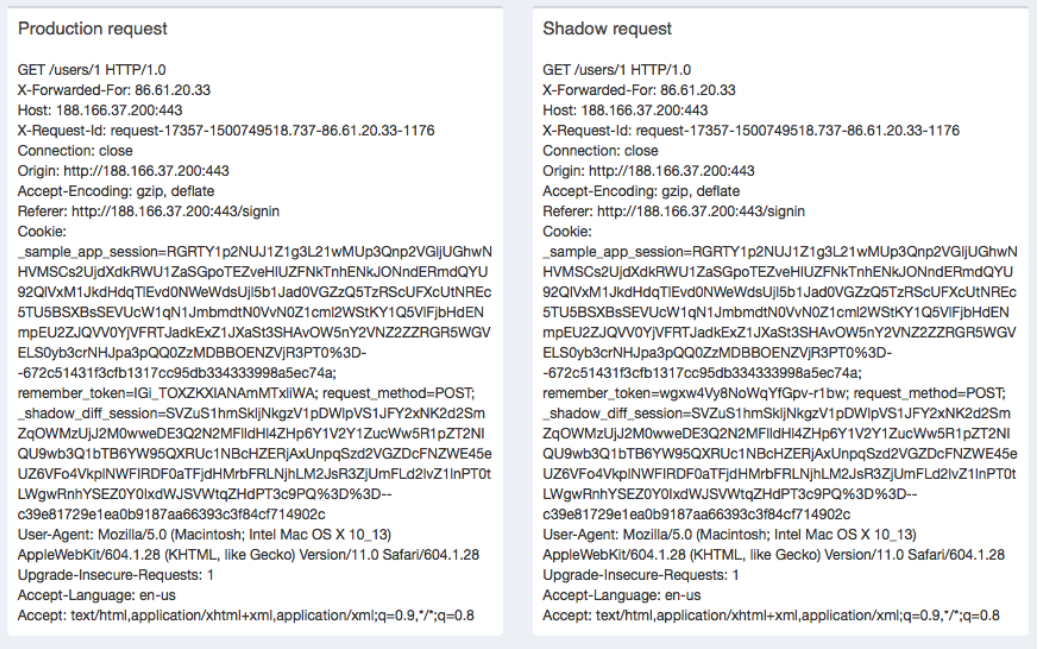
Slika 3.5: Zahteve, ki so v okviru testiranja dostopale do spletne poti `/signin` in vsote točkovnih ocen sprememb v odgovorih na podlagi ustreznih pravil za zaznavanje regresij.

prikazuje slika 3.9. Prikazana so samo tista pravila, za katera smo v spletnih odgovorih našli ujemanje s pripadajočimi regularnimi izrazi. Poleg imen, tipov pravil in številčnih ocen sprememb je prikazana tudi vsebina spletnih odgovorov obeh strežnikov, ki se je ujemala z regularnim izrazom. V primeru ujemanja s pravilom za zaznavanje vsebinske regresije se celotna vrstica tabele s pravili obarva rdeče.

- *Primerjavo aplikacijskih metrik v produkcijskem in senčenem okolju.* V primeru, da vrednost vsaj ene od aplikacijskih metrik iz senčenega okolja preseže dovoljeno odstopanje, se celoten razdelek obarva rdeče, kot je razvidno iz slike 3.10.

3.6.6 Ročna ponastavitev negativnih ocen

Razlike med odgovoroma produkcijskega aplikacijskega strežnika in senčenega aplikacijskega strežnika s testirano izdajo niso nujno znak regresije. Če ignoriramo podatkovni šum, so razlike v spletnih odgovorih pogosto posledica namerne spremembe izvorne kode spletne aplikacije ali pa izrednega dogodka (npr. vrednost dane aplikacijske metrike je nenavadno visoka pri odgovoru na eno samo zahtevo zaradi zasičenosti omrežja ali obremenitve strežnika). Zato smo v spletno aplikacijo za nadzor in analizo senčenja dodali možnost ročnega ponastavljanja negativnih ocen, kar pomeni nastavitev številčne ocene dane spremembe v odgovorih na nič. To funkcionalnost lahko razvijalec uporabi, če presodi, da negativna ocena ne more biti posledica regresije. Ocene posameznih sprememb odgovorov strežnikov lahko ponastavimo bodisi za konkretne spletne zahteve bodisi za vse zahteve, ki dostopajo do dane spletne poti.



Slika 3.6: Spletna zahteva za produkcijski (levo) in senčeni strežnik (desno). Slika je bila zajeta v pasivnem načinu delovanja posredniškega strežnika za senčenje, zato sta zahtevi v prikazanem primeru identični.

3.7 Povezava s storitvijo GitHub

Da bi bila razvita rešitev za avtomatsko zaznavanje regresij združljiva z uveljavljenim delovnim tokom razvoja programske kode, smo jo povezali z zahtevami za pregled (angl. *pull request*) storitve GitHub.

3.7.1 Zagon in ustavitev avtomatskega zaznavanja regresij

Spletni portal GitHub omogoča, da ob določenem dogodku, ki izvira iz izbrane zahteve za pregled na portalu, o dogodku obvesti zunanjo storitev, pri čemer uporabi mehanizem *web hook*. Zanimali so nas predvsem dogodki, povezani s spremembami oznake (angl. *label*) zahteve za pregled. V zahtevi za pregled ciljne spletne aplikacije, s pomočjo katere želimo integrirati našo rešitev za avtomatsko zaznavanje regresij, moramo dodati oznako `duplicate`. V primeru, da razvijalec k zahtevi za pregled doda omenjeno oznako, GitHub o dogodku obvesti spletno aplikacijo za analizo in nadzor senčenja, ki v nadaljevanju

Slika 3.7: Primer spletnega odgovora produkcijskega (levo) in senčenega strežnika (desno). Iz prikaza so jasno razvidne razlike med odgovoroma.

sproži postopek senčenja. Obenem GitHub spletni aplikaciji za analizo in nadzor senčenja pošlje tudi podatek o zgoščeni vrednosti uveljavitve spremembe, avtorja zahteve za pregled in spletno povezavo do strani z zahtevo za pregled. Nasprotno pa odstranitev oznake `duplicate` prekine postopek senčenja, če se slednji še ni zaključil. Dodajanje oznake `duplicate` k zahtevi za pregled na portalu GitHub prikazuje slika 3.11.

3.7.2 Sporočanje rezultatov

Po končanem postopku avtomatskega zaznavanja in analize regresij naša rešitev obvesti portal GitHub o rezultatih, ki si jih nato lahko ogledamo na spletni strani zahteve za pregled. Na ta način lahko razvijalci takoj vidijo, ali je korak testiranja z uporabo senčenja uspel ali ne (tj. ali smo v novi izdaji zaznali regresije ali ne), prav tako pa lahko na spletni strani zahteve za pregled poiščejo povezavo do nadzorne plošče naše rešitve, kjer lahko pridobijo podrobne informacije o izvedenih analizah in rezultatih.

Primer prikaza rezultatov testiranja nove izdaje na spletni strani zahteve za pregled preko portala GitHub prikazuje slika 3.12.

Active rules

Rule name	Rule action	Score modifier	Regex	Add rule
ETag identifier	modify	0.0	ETag	Remove
Cookie field	modify	0.0	Set-Cookie	Remove
Total runtime metric	modify	0.0	X-Runtime	Remove
CSRF header token	modify	0.0	<meta content=(.)* name="csrf-token" />	Remove
CSRF form token	modify	0.0	<form accept-charset=\"UTF-8\" action=\"/sessions\" method=\"post\"><div style=\"margin:0;padding:0;display:inline\"> <input name=\"utf8\" type=\"hidden\" value=\"✓\" /><input name=\"authenticity_token\" type=\"hidden\" value=\"(.*?)\" />	Remove
View runtime metric	modify	0.0	X-View-Runtime	Remove
Databse runtime metric	modify	0.0	X-Db-Runtime	Remove
SQL queries metric	modify	0.0	X-Sql-Queries	Remove

Slika 3.8: Vsa pravila za ocenjevanje vsebinskih razlik v spletnih odgovorih, ki pa zaradi neujemanja z regularnim izrazom niso nujno uporabljena pri analizi odgovorov na dano spletno zahtevo.

Applied rules

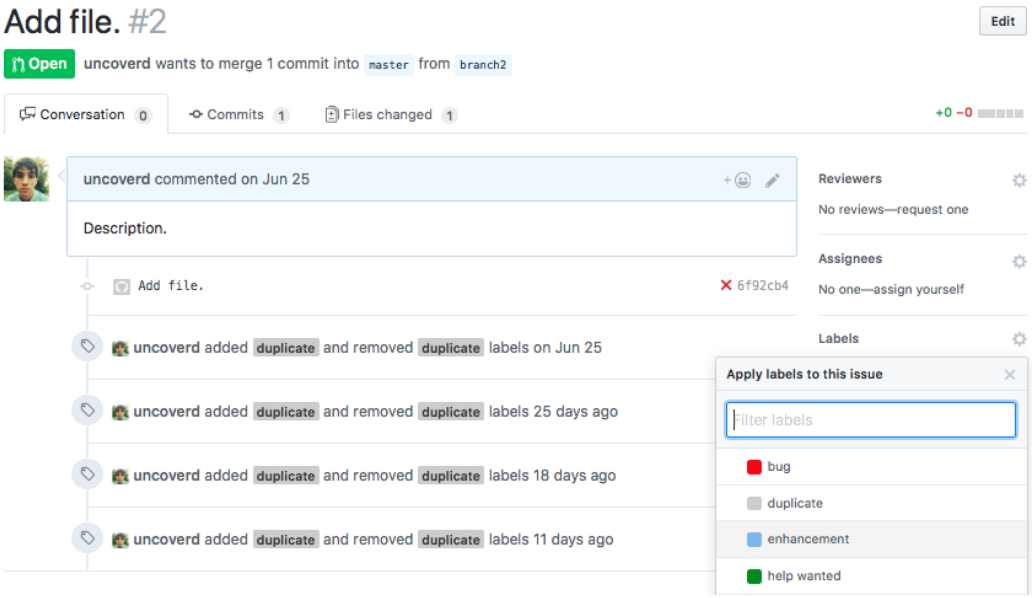
Rule name	Rule action	Line score	Line content
Cookie field	modify	0.0	-Set-Cookie: _sample_app_session=T0lSY2d0VXhZYzJRV2NtRXUxVnBYUTdOQWZmd0kyZTVFaTc3RWFIRXU3c0FYZWROSWxsYys5a0R5MFZMR3RHUEErYTVCZ3ZGS1VrZERXQjlUblEL0hDVzdOdWJ5SC9jekFyTVlKS0hGWDdPaHl6cnZldGdTTGNEeWFYbVdkdEQzOGFkbDR1TnFrVlVvS2liMTFPYnhTdXA2Z0l; path=/; HttpOnly +Set-Cookie: _sample_app_session=dzhDQkhtYWcxTVdsd2ROOEk5a3ZVRFBVc0tkbnNFK1BOcjlGNSthYWdTVkV3ZzJtajlGbitEKzN0SDR4WitoQytRc3A5amJGUHJubnZvV216aThyK3RYMWE1aTlaQlpUcjQ2QklwOHdVS0RVWmVtaUptc29rekxPdTY5YUVpTVBEZU13N2pseHJKam5aNEZ3KzhmaDJOaTRERkJ; path=/; HttpOnly
Total runtime metric	modify	0.0	-X-Runtime: 0.087888 +X-Runtime: 0.090923
View runtime metric	modify	0.0	-X-View-Runtime: 46 +X-View-Runtime: 50
Databse runtime metric	modify	0.0	-X-Db-Runtime: 35 +X-Db-Runtime: 33
Response metrics	modify	-20.0	

Slika 3.9: Uporabljena pravila za ocenjevanje vsebinskih razlik v spletnih odgovorih, tj. tista, za katera smo našli ujemanje s pripadajočimi regularnimi izrazi.

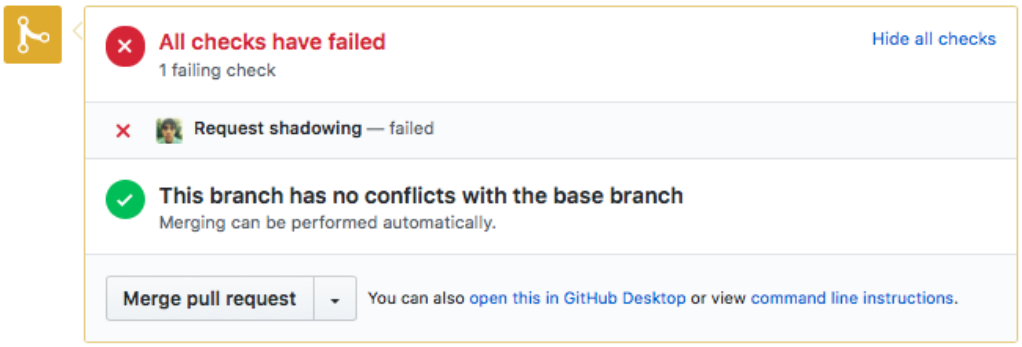
Shadow response metrics

-22.0 ms	+ 264.0 ms	No change
DATABASE RUNTIME	VIEW RUNTIME	SQL QUERIES

Slika 3.10: Primerjava aplikacijskih metrik iz odgovorov produkcijskega in senčenega strežnika na dano spletno zahtevo.



Slika 3.11: Dodajanje oznake `duplicate` k zahtevi za pregled na portalu GitHub.



Slika 3.12: Pregled rezulatov senčenja na spletni strani zahteve za pregled.

Poglavje 4

Integracija rešitve s postavitvenim cevovodom ciljne aplikacije

V pričujočem poglavju opišemo postopek integracije spletnih aplikacij, izdelanih z ogrodjem Ruby on Rails, z razvito rešitvijo za avtomatsko zaznavanje regresij. Postopek integracije demonstriramo na konkretnem primeru; v razdelku 4.1 najprej opišemo spletno aplikacijo, s katero smo integrirali našo rešitev, in predstavimo njen cevovod zvezne postavitve pred začetkom integracije. V razdelkih 4.2 in 4.3 pa predstavimo prilagoditve ciljne spletne aplikacije in njenega cevovoda zvezne postavitve, ki jih zahteva integracija z našo rešitvijo.

4.1 Ciljna aplikacija

Aplikacija, ki smo jo izbrali za integracijo z našo rešitvijo, je namenjena obveščanju o prometnih dogodkih na izbranih geografskih območjih v realnem času. Končnim uporabnikom pošilja obvestila o zastojih in drugih izrednih dogodkih na cestah, ki jih pogosto uporabljajo. Izdelana je v programskem jeziku Ruby s pomočjo ogrodja za izgradnjo spletnih aplikacij Ruby on Rails, na voljo pa je tako preko spletnega vmesnika kot tudi v obliki mobilne aplikacije za platformo iOS. Izbrali smo jo iz dveh razlogov:

1.) ker smo jo dobro poznali že pred začetkom razvoja naše programske rešitve za avtomatsko zaznavanje regresij,

2.) ker že ima implementiran cevovod zvezne postavitve, ki omogoča avtomatsko postavitev nove izdaje v produkcijsko okolje.

Konfiguracija strežnikov v opisani aplikaciji je avtomatizirana in se izvede v približno desetih minutah.

Na tem mestu omenimo, da je postopek integracije z našo rešitvijo enak za vse spletne aplikacije Ruby on Rails, ki že imajo cevovod zvezne postavitve. V preostanku pričujočega poglavja omenjeni postopek predstavimo na primeru izbrane ciljne aplikacije predvsem zaradi lažjega razumevanja.

4.1.1 Izhodiščni cevovod zvezne postavitve

Cevovod zvezne postavitve ciljne aplikacije vsebuje deset korakov, ki se izvajajo s pomočjo storitve za zvezno integracijo Codeship [5]. Ob vsaki uveljavitvi spremembe v glavni veji repozitorja izvorne kode, se izvede naslednje zaporedje korakov:

Korak 1. Na oddaljenem strežniku se pripravi izvajalno okolje za storitev Codeship, kar vključuje izbiro verzije programskega jezika Ruby, namestitev potrebnih programskih paketov in prenos izvorne kode.

Sledi testiranje izdaje, v katerem se izvedejo različni načini testiranja. Vsak sklop testiranja je časovno bolj potraten od predhodnega, hkrati pa razvijalcem nudi višjo stopnjo zaupanja v pravilnost testirane izdaje:

Korak 2. Izvedejo se testi enot.

Korak 3. Izvedejo se integracijski testi.

Korak 4. Izvedejo se sprejemni testi.

Korak 5. Izvede se skripta za zaznavanje nevarnih podatkovnih migracij v izvorni kodi.

V primeru, da se vsi zgornji testi uspešno zaključijo (tj. brez napak), se sproži postopek postavitve in testiranja nove izdaje spletne aplikacije, ki poteka v dveh aplikacijskih okoljih - vmesnem in produkcijskem.

Korak 6. Izdaja se postavi v vmesno okolje.

Korak 7. Delovanje spletne aplikacije v vmesnem okolju preverimo s pomočjo dimnih testov, ki simulirajo interakcijo končnega uporabnika s spletno aplikacijo.

Če v vmesnem okolju ne zaznamo napak, se zgornje izvede še za produkcijsko okolje:

Korak 8. Izdaja se postavi v produkcijsko okolje.

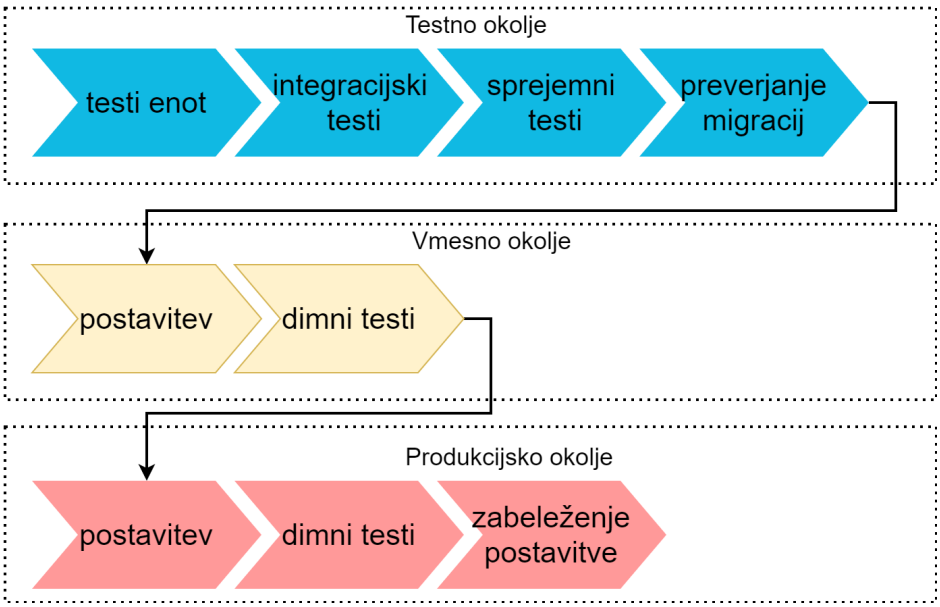
Korak 9. Delovanje spletne aplikacije v produkcijskem okolju preverimo s pomočjo dimnih testov, ki simulirajo interakcijo končnega uporabnika s spletno aplikacijo.

Zadnja stopnja cevovoda zvezne postavitve aplikacije, ki smo jo izbrali za integracijo z našo rešitvijo za avtomatsko zaznavanje regresij, pa skrbi za beleženje:

Korak 10. Spletna storitev za beleženje delovanja aplikacij New Relic [28] zabeleži uspešno postavitev aplikacije.

V primeru, da razvijalec spremembo izvorne kode uveljavi v pomožni razvojni veji (tj. ne glavni), in naredi zahtevo za pregled (katere cilj je uveljaviti spremembe v glavni veji), storitev Codeship izvede samo prvih pet korakov postavitvenega cevovoda. Razvijalci si lahko nato rezultate testiranja ogledajo na spletni strani zahteve za pregled.

Opisane korake izhodiščnega cevovoda zvezne postavitve izbrane spletne aplikacije grafično ponazarja slika 4.1.



Slika 4.1: Izhodiščni cevovod zvezne postavitve spletne aplikacije, s katero smo v nadaljevanju integrirali našo rešitev za avtomatsko zaznavanje regresij.

4.2 Prilagoditev aplikacije

V nadaljevanju opišemo korake, potrebne za integracijo naše rešitve s ciljnimi aplikacijami, ki uporabljajo spletno ogrodje Ruby on Rails in sistem za upravljanje podatkovnih

baz PostgreSQL. Integracija naše rešitve s ciljnimi aplikacijami, ki uporabljajo druga spletna ogrodja, zahteva prilagoditve opisanih korakov, ki so odvisne od izbranega spletnega ogrodja.

4.2.1 Vključitev knjižnice za sporočanje aplikacijskih metrik

Da lahko razvita rešitev izvede avtomatsko zaznavanje regresij, kot vhodne podatke potrebuje tudi aplikacijske metrike ciljne aplikacije. V ta namen moramo v ciljno spletno aplikacijo vključiti knjižnico za sporočanje aplikacijskih metrik, predstavljeno v razdelku 3.5. Slednje na nivoju ciljne spletne aplikacije zahteva dve spremembi:

1.) V standardno datoteko aplikacij Ruby on Rails, v kateri navedemo knjižnice, ki jih spletna aplikacija uporablja (t. i. *Gemfile*), dodamo razvito knjižnico za sporočanje aplikacijskih metrik na spodnji način:

```
gem 'server_timing_middleware',
:git => 'https://github.com/uncoverd/server_timing_middleware.git'.
```

2.) Sporočanje aplikacijskih metrik nato aktiviramo tako, da v prav tako standardno datoteko aplikacij Ruby on Rails, `application.rb`, dodamo vrstico

```
config.middleware.use Rack::ServerTimingMiddleware.
```

Izvedba zgornjih dveh korakov zadošča, da lahko ciljna aplikacija razviti rešitvi za avtomatsko zaznavanje regresij sporoča aplikacijske metrike.

4.2.2 Ponastavitev zaporedja identifikatorjev senčene podatkovne baze

Ob replikaciji podatkov iz podatkovne baze v produkcijskem okolju v podatkovno bazo v senčenem okolju se lahko sproži izjema zaradi kršitve unikatnosti primarnih ključev. Sistem za upravljanje podatkovnih baz PostgreSQL za generiranje naraščajočih polj uporablja koncept zaporedij (angl. *sequences*), ki se hranijo ločeno od tabele, v kateri so uporabljeni. V primeru, da replikacija ne vsebuje tabel z zaporedji, je zato lahko izračunano naraščajoče polje enako že obstoječemu zapisu v podatkovni bazi. V primeru, da ne želimo replicirati tabel z zaporedji lahko namesto tega v ciljni aplikaciji ponastavimo zaporedje z enostavnim blokom programske kode Ruby:

```
ActiveRecord::Base.connection.tables.each do |t|
    ActiveRecord::Base.connection.reset_pk_sequence!(t)
end.
```

4.2.3 Preprečevanje klicev zunanjih storitev v senčenem okolju

V primeru, da ciljna spletna aplikacija uporablja zunanje storitve (npr. kliče zunanje programske vmesnike REST), uporaba naše rešitve zahteva nekaj pazljivosti. Pri komunikaciji z zunanjimi storitvami so problematične zgolj nevarne zahteve (tokrat govorimo o varnosti zahtev, ki jih ciljna spletna aplikacija pošilja zunanjim storitvam in ne končni uporabnik ciljni aplikaciji), saj bi lahko prišlo do neželenih posledic zaradi dvojne obdelave zahteve v produkcijskem in senčenem okolju. Na primer, ciljna spletna aplikacija lahko za zaračunavanje plačljivih funkcionalnosti uporablja zunanjo storitev. Če uporabnikovo zahtevo obdeluje senčeni aplikacijski strežnik, tj. če aplikacija teče v senčenem okolju, je potrebno onemogočiti klic zunanje storitve, saj bi v nasprotnem primeru zunanja storitev uporabnikov račun bremenila dvakrat.

Ključnega pomena je torej, da pred izvedbo klicev, ki spreminjajo stanje zunanje storitve, ciljna spletna aplikacija preveri, ali teče v produkcijskem ali v senčenem okolju. Slednje lahko enostavno nadziramo s pomočjo okoljske spremenljivke, katere vrednost aplikacija preveri pred nevarnim klicem zunanje storitve. V primeru, da ciljna aplikacija teče v senčenem okolju, mora torej onemogočiti klic zunanje storitve. Če rezultat nevarnega klica zunanje storitve potrebujemo, npr. ker ga želimo vključiti v spletni odgovor, kot rezultat klica zunanje storitve uporabimo prednastavljeno vrednost. V tem primeru moramo za natančnejšo analizo spletnih odgovorov s strani rešitve za zaznavanje regresij dodati pravilo, ki dotično vsebino odgovora na spletno zahtevo izključi iz analize.

V ciljni spletni aplikaciji smo v senčenem okolju preprečili klic zunanje storitve za pošiljanje potisnih obvestil, saj bi uporabnik v nasprotnem primeru obvestilo prejel dvakrat. Klicev zunanje storitve za pridobivanje prometnih podatkov v senčenem okolju nismo preprečili, saj so slednji varni in se torej lahko izvajajo v obeh okoljih.

4.2.4 Nastavitev šifrirnega ključa `secret_key_base`

Novejše verzije ogrodja Rails (od verzije št. 4 naprej) podatke o uporabniški seji privzeto shranjujejo v kriptirane piškotke. Zaradi varnostnih razlogov aplikacijski strežnik piškotke šifrira s šifrirnim algoritmom AES-256. Algoritem AES temelji na simetrični kriptografiji, zato se za šifriranje čistopisa in odšifriranje tajnopisa uporablja isti skrivni ključ. Ogrodje

Rails omenjeni ključ uporablja tako za šifriranje sejnih žetonov kot tudi za ustvarjanje in preverjanje žetonov CSRF. Skrivni ključ moramo navesti v standardni datoteki za shranjevanje občutljivih podatkov aplikacij Ruby on Rails, tj. `config/secrets.yml`.

Pri uporabi senčenja lahko posamezno spletno zahtevo obdelujeta aplikacijska strežnika v dveh različnih okoljih. Omenjeni ključ mora zato biti enak za vse instance aplikacijskih strežnikov v produkcijskem in senčenem okolju, saj v nasprotnem primeru lahko pride do napak ob oddaji vnosnih obrazcev, ki zahtevajo preverjanje žetona CSRF.

4.3 Prilagoditev cevovoda zvezne postavitve

V sklopu integracije naše rešitve za avtomatsko zaznavanje regresij moramo po prilagoditvi ciljne spletne aplikacije spremeniti še njen cevovod zvezne postavitve. Potrebna je zgolj ena sprememba: v cevovod zvezne postavitve moramo po koncu sklopa testiranja dodati en sam korak, in sicer *avtomatsko postavitev izbrane izdaje v senčeno okolje*. Ta korak se izvede na enak način kot postavitev nove izdaje aplikacije v vmesno in produkcijsko okolje. S tem je novejša različica aplikacije, tj. različica, ki jo testiramo v okviru zahteve za pregled, pripravljena na senčenje. Postopek zaznavanja regresij na podlagi senčenja zahtev nato ročno sproži razvijalec, rezultati senčenja pa so prikazani na spletni strani storitve GitHub.

Na tem mestu poudarimo, da se senčenje izvede samo v primeru uveljavitev sprememb pri zahtevah za pregled iz ločenih razvojnih vej na glavno vejo, tj. preden uveljavimo spremembe v glavni veji. Pri uveljavitvi sprememb v glavni veji cevovod zvezne postavitve ostane popolnoma enak, saj uveljavitev sprememb v tem primeru pomeni prehod na novo izdajo aplikacije v produkciji.

Integracija naše rešitve z opisano ciljno aplikacijo torej zahteva dodatni korak za korakom 5 iz razdelka 4.1.1.

Poglavje 5

Ovrednotenje

Po končani integraciji s ciljno spletno aplikacijo smo razvito rešitev za avtomatsko zaznavanje in analizo regresij še ovrednotili. V pričujočem poglavju najprej na kratko predstavimo kriterije vrednotenja, ki jih v nadaljevanju uporabimo za ovrednotenje implementirane rešitve z različnih vidikov. Našo rešitev nato še primerjamo z edino nam znano primerljivo rešitvijo, Diffy.

5.1 Kriteriji vrednotenja

V nadaljevanju navajamo nabor kriterijev, ki smo jih uporabili za vrednotenje naše rešitve. Na kratko predstavimo tudi njihovo pomembnost v kontekstu razvoja, postavitve ali uporabe spletnih aplikacij.

- **Zahtevnost integracije s ciljno aplikacijo** Enostavna integracija rešitve s ciljno spletno aplikacijo omogoča minimalen vpliv na hitrost razvoja aplikacije. Zahtevnost integracije rešitve pogosto vpliva na odločitev skupine razvijalcev za oziroma proti uvedbi novega načina testiranja.
- **Dodatna infrastruktura** Količina dodatne infrastrukture, potrebne za uvedbo novega načina testiranja, vpliva predvsem na povečanje stroškov. Načini zaznavanja regresij, ki zahtevajo več dodatne infrastrukture (npr. modro-zelene postavitve) so v splošnem dražji in posledično manj priljubljeni kot načini zaznavanja regresij, ki zahtevajo manj dodatne infrastrukture (npr. kanarske izdaje). Poleg tega se s povečevanjem dodatne infrastrukture prav tako povečuje tudi količina režijskega dela, potrebnega za njeno konfiguracijo in vzdrževanje.
- **Količina ročnega dela** Na uspešnost uvedbe novega načina testiranja izdaj v

razvijalski proces močno vpliva tudi količina zahtevanega sprotnega ročnega dela razvijalcev. Če nov način testiranja od razvijalcev ne zahteva veliko dodatnega dela in obenem prinaša veliko korist, jo bodo slednji bolj verjetno uporabljali.

- **Transparentnost za končne uporabnike** Transparentnost preverjanja delovanja nove izdaje je eden od ključnih kriterijev pri izbiri metode zaznavanja regresij v produkcijskem okolju. Če imajo mehanizmi za podporo testiranja velik doprinos k odzivnemu času aplikacije, ali če regresije testirane izdaje vplivajo na končne uporabnike, ima uvedba rešitve negativen vpliv na uporabniško izkušnjo.
- **Zaznani tipi regresij** Koristnost posameznega načina testiranja nove izdaje je odvisna predvsem od tipa regresij, ki jih slednji lahko zazna. Uvedba novega načina testiranja ciljne aplikacije je v večini primerov smiselna le, če slednji omogoča zaznavanje novih tipov regresij, ki jih obstoječi načini testiranja ne omogočajo.
- **Nivo zaupanja v pravilnost izdaje** Na nivo zaupanja v pravilnost izdaje s strani razvijalcev vpliva predvsem celovitost preverjanja izdaje s pomočjo izbranega načina testiranja.

5.2 Zahtevnost integracije s ciljno aplikacijo

Čeprav je težavnost integracije razvite rešitve za avtomatsko zaznavanje in analizo regresij z izbrano spletno aplikacijo odvisna tudi od aplikacije same, v splošnem ne zahteva veliko časa. Večina sprememb, potrebnih za integracijo senčenja produkcijskih zahtev s ciljno aplikacijo, je preprostih in od razvijalca ne zahteva veliko časa. Za izvedbo vseh zahtevanih prilagoditev ciljne spletne aplikacije, opisanih v razdelku 4.2, smo porabili približno eno uro. Od tega smo največ časa porabili za prilagoditve obnašanja spletne aplikacije v senčenem okolju zaradi komunikacije z zunanjimi storitvami.

5.3 Dodatna infrastruktura

Količina dodatne infrastrukture, ki jo zahteva uvedba naše rešitve, je odvisna tako od izbranega načina delovanja posredniškega strežnika za senčenje, kot tudi od deleža zahtev, ki jih želimo senčiti, s tem pa nenazadnje tudi od obremenitve ciljne aplikacije.

Naša rešitev predvideva dodatni strežnik, na katerega namestimo spletno aplikacijo za analizo in nadzor senčenja, podatkovno zbirko Redis ter posredniški strežnik za senčenje. Poleg tega v primeru, da želimo posredniški strežnik za senčenje uporabljati samo v pasivnem načinu delovanja (s tem pa analizirati zgolj varne zahteve), potrebujemo samo en dodatni aplikacijski strežnik in eno dodatno podatkovno bazo za senčeno okolje.

Če pa želimo posredniški strežnik za senčenje uporabljati v aktivnem načinu, potem za vsak aplikacijski strežnik v senčenem okolju potrebujemo dodatno podatkovno bazo. Kot rečeno v razdelku 3.2.2, lahko z nastavitvijo uteži posameznih aplikacijskih strežnikov poljubno spreminjamo delež senčenih zahtev; v primeru, da število zahtev, ki jih želimo senčiti, presega zmogljivosti enega aplikacijskega strežnika, v senčeno okolje lahko dodamo poljubno število aplikacijskih strežnikov. Vendar pa moramo za vsak senčeni aplikacijski strežnik namestiti ločeno instanco posredniškega strežnika za senčenje. Zaradi nizke porabe sistemskih virov lahko več instanc posredniškega strežnika za senčenje namestimo kar na isti strežnik kot spletno aplikacijo za nadzor in analizo senčenja.

Čas za konfiguracijo in postavitev aplikacijskih strežnikov, podatkovnih baz, replikacije podatkovne baze in posredniškega strežnika za senčenje je zelo odvisen od obstoječega pristopa k postavitvi spletne aplikacije, s katero integriramo predstavljeno rešitev za avtomatsko zaznavanje in analizo regresij. V primeru ciljne aplikacije, opisane v razdelku 4.1, je bila postavitev aplikacijskih in podatkovnih strežnikov povsem avtomatizirana, zato smo se odločili avtomatizirati tudi konfiguracijo in postavitev dodatne infrastrukture. V ta namen smo porabili nekaj dni, v našem primeru predvsem na račun avtomatizacije postopka. Največ časa smo vložili predvsem v konfiguracijo sistema Bucardo in vzpostavitev replikacije podatkovne baze.

5.4 Količina ročnega dela

Razvijalec mora vsako izdajo, ki jo želi testirati s pomočjo naše rešitve za zaznavanje regresij s pomočjo senčenja produkcijskih zahtev, najprej sam izbrati preko portala GitHub na spletni strani pripadajoče zahteve za pregled, kot je opisano v poglavju 3.7.

Nova izdaja ciljne spletne aplikacije lahko načrtno predvidi spremembo v spletnem odgovoru na vsaj eno zahtevo glede na produkcijsko verzijo. V tem primeru mora razvijalec za pravilno analizo regresij preko spletne aplikacije za analizo in nadzor senčenja dodati vsaj eno pripadajoče pravilo, ki pričakovano spremembo izloči iz analize odgovorov.

Po končani analizi je rezultat testiranja lahko negativen. V tem primeru razvijalec preko spletne aplikacije za analizo in nadzor senčenja preveri zaznane regresije in ustrezno popravi izvorno kodo spletne aplikacije. Po uveljavitvi sprememb potisne novo različico aplikacije v isto zahtevo za pregled, kar ponovno sproži postopek testiranja s senčenjem.

Ročno posredovanje razvijalca z razvito rešitvijo za avtomatsko zaznavanje regresij v primeru pozitivnega rezultata testiranja ne predvideva več kot le nekaj minut za posamezno izdajo. Drugi obstoječi načini testiranja novih izdaj v produkcijskem okolju, kot so na primer modro-zelene postavitve in kanarske izdaje, za razliko od naše rešitve zahtevajo aktivno in dolgotrajno preverjanje pravilnega delovanja nove izdaje s strani razvijalcev.

Čas, potreben za senčenje želenega števila spletnih zahtev, je odvisen od števila vseh spletnih zahtev za ciljno aplikacijo. Če ciljna spletna aplikacija prejema relativno malo uporabniških zahtev, lahko postopek zaznavanja regresij v novi izdaji traja dolgo, predvsem na račun zbiranja zadostnega števila spletnih zahtev s strani posredniškega strežnika za senčenje.

5.5 Transparentnost za končne uporabnike

Delovanje predstavljene rešitve za avtomatsko zaznavanje in analizo regresij je za končnega uporabnika ciljne spletne aplikacije povsem transparentno. Dodatno procesiranje spletnih zahtev zaradi uporabe posredniškega strežnika za senčenje zahteva zgolj nekaj milisekund, kar je z vidika odzivnega časa ciljne aplikacije zanemarljivo. Poleg tega končni uporabnik vedno vidi samo odgovor produkcijskega aplikacijskega strežnika, tako da morebitne regresije v testirani izdaji ne vplivajo na pravilnost delovanja ciljne spletne aplikacije.

5.6 Zaznani tipi regresij

Analiza odgovorov senčenega in produkcijskega strežnika na produkcijske zahteve omogoča avtomatsko zaznavanje vsebinskih in zmogljivostnih regresij v produkcijskem okolju, ki jih uveljavljeni načini testiranja (npr. testi enot in integracijski testi), ki se izvajajo v testnem okolju, ne morejo zaznati.

5.6.1 Zmogljivostne regresije

Aplikacijska strežnika v senčenem in produkcijskem okolju, ki prejemata zahteve posredniškega strežnika za senčenje, obdelujeta enake zahteve, zato lahko neposredno primerjamo njune zmogljivostne metrike iz pripadajočih spletnih odgovorov. V aktivnem načinu delovanja posredniškega strežnika za senčenje sta namreč oba aplikacijska strežnika enakomerno obremenjena, tako da je negativni rezultat analize aplikacijskih metrik, ki je posledica prevelikih odstopanj od pričakovanih vrednosti, zelo verjetno posledica zmogljivostne regresije.

V ciljni spletni aplikaciji lahko s pomočjo naše rešitve zaznamo zmogljivostne regresije, ki nastanejo kot posledica:

- uvedbe zahtevnejših izračunov za prikaz zahteve,
- nepričakovano velikega povečanja števila poizvedb za podatkovno bazo,
- uvedbe zahtevnejših poizvedb za podatkovno bazo,

- napačnih nastavitev upravljanja predpomnilnikov,
- zmogljivostnih regresij spletnega ogrodja in zunanjih programskih knjižnic, uporabljenih v ciljni spletni aplikaciji.

5.6.2 Vsebinske regresije

Naša rešitev za avtomatsko zaznavanje in analizo regresij je najbolj koristna za odkrivanje sprememb v spletnih odgovorih, ki nakazujejo na vsebinske regresije funkcionalnosti z nepopolnimi nabori obstoječih testov. Zaznamo lahko kopico vsebinskih regresij, ki jih razvijalci niso predvideli, ampak se v produkcijskem okolju še vseeno pojavljajo. V ciljni spletni aplikaciji lahko s pomočjo naše rešitve zaznamo vsebinske regresije, ki nastanejo kot posledica:

- napačne obravnave robnih pogojev v aplikacijski logiki,
- napačne obravnave kodiranj vnešenih podatkov,
- nepopolnega preurejanja (angl. *refactoring*) funkcionalnosti v spletni aplikaciji,
- nepričakovanih sprememb v delovanju zunanjih knjižnic,
- nepričakovanih sprememb v delovanju spletnega ogrodja.

Obenem analiza vsebinskih regresij razvijalcu da zagotovilo, da njegova sprememba izvorne kode nima neželenih učinkov na druge funkcionalnosti ciljne aplikacije. Razvijalec lahko namreč podrobno pregleda rezultate analize spletnih odgovorov na senčene zahteve za posamezne spletne poti in se prepriča, da njegova sprememba izvorne kode učinkuje samo na dano funkcionalnost.

5.7 Nivo zaupanja v pravilnost izdaje

Ali se bo nivo zaupanja v pravilnost nove izdaje s strani razvijalcev po uvedbi naše rešitve povečal, je odvisno od števila analiziranih odgovorov in kvalitete uporabljenih pravil za izločanje šuma. Če senčimo premajhno število spletnih zahtev, katerih obdelava proži zgolj majhen del izvajalnih poti nove izdaje, je zelo verjetno, da morebitnih regresij ne bomo zaznali. Prav tako uporaba preveč splošnih pravil za izločanje šuma lahko povzroči, da regresije napačno klasificiramo kot šum.

Ustrezno konfigurirano senčenje produkcijskih zahtev nam je v sklopu ciljne aplikacije iz razdelka 4.1 prineslo veliko povečanje zaupanja v pravilnost novih izdaj. Uporaba produkcijskih zahtev za preverjanje pravilnosti nove izdaje sicer zahteva več časa, vendar vrne

bolj zanesljiv rezultat testiranja izdaje kot nižjenivojske oblike avtomatskega testiranja (na primer testi enot), s tem pa jih dobro dopolnjuje.

Za razliko od ostalih načinov testiranja v postavitvenem cevovodu, ki uporabljajo vhodne podatke, določene s strani razvijalcev, uporaba naše rešitve za senčenje produkcijskih zahtev namreč omogoča analizo veliko večjega števila nepredvidenih robnih pogojev in vzorcev uporabe ciljne aplikacije. Razvijalcev namreč po implementaciji nove izdaje ne zanima njeno delovanje v simuliranih pogojih, ampak v produkcijskem okolju.

5.8 Primerjava z obstoječo rešitvijo Diffy

Odprtokodna programska rešitev Diffy je bila razvita s strani razvijalcev podjetja Twitter, in sicer za potrebe testiranja njihovih spletnih aplikacij. Rešitev Diffy je v prvi vrsti namenjena odkrivanju vsebinskih regresij v ciljnih aplikacijah, za razliko od naše rešitve, ki omogoča tudi odkrivanje zmogljivostnih regresij.

Podobno kot naša rešitev, Diffy spletno zahtevo posreduje v senčeno in produkcijsko okolje. Vendar pa Diffy za razliko od naše rešitve predvideva uporabo najmanj dveh produkcijskih aplikacijskih strežnikov (in ne najmanj enega kot naša rešitev) in senčenega aplikacijskega strežnika.

Za razliko od naše rešitve, ki šum v spletnih odgovorih zaznava in izloča s pomočjo množice pravil, Diffy šum izloča s primerjavo deležev sprememb v odgovorih vseh treh instanc aplikacijskih strežnikov. Najprej izvede dve primerjavi spletnih odgovorov - med dvema produkcijskima aplikacijskima strežnikoma in enim produkcijskim ter enim senčenim aplikacijskim strežnikom. Spremembe, ki se pojavljajo v odgovorih dveh produkcijskih aplikacijskih strežnikov, Diffy klasificira kot šum, saj se na obeh strežnikih izvaja ista (tj. trenutna) izdaja ciljne aplikacije, ki jo smatramo kot pravilno. Če so deleži sprememb v odgovorih približno enaki za par dveh produkcijskih aplikacijskih strežnikov in za par produkcijskega ter senčenega aplikacijskega strežnika, potem jih Diffy tudi v testirani izdaji klasificira kot šum in izloči iz analize.

Razlike v odgovorih produkcijskega in senečenga aplikacijskega strežnika, ki ostanejo, so tako lahko le posledica regresije nove izdaje. Opisan pristop za razliko od našega sistema uporabniško definiranih pravil omogoča bolj avtomatizirano analizo zahtev in njihovih odgovorov.

Diffy zaradi povsem avtomatskega zaznavanja šuma omogoča zmanjšanje števila napačno zaznanih regresij v primerjavi z našo rešitvijo. Posledično od razvijalcev zahteva tudi manj dela. Dodatna prednost je tudi zmanjšanje dela z vzdrževanjem seznama privzetih pravil med posameznimi izdajami. V naši rešitvi lahko namreč pravila, ki so bila aktualna v predhodni izdaji in jih ne odstranimo iz sistema, v prihodnjih izdajah zakrijejo morebitne

regresije.

Glavna prednost sistema za zaznavanje šuma s pravili, zaradi katere smo se odločili za njegovo implementacijo, je sposobnost senčenja nevarnih zahtev. Uporaba dveh instanc aplikacijskih strežnikov na način, kot ga implementira rešitev Diffy, v produkcijskem okolju, za obdelavo iste zahteve namreč ni možna. Istočasno vnašanje ali spreminjanje istih zapisov v produkcijski podatkovni bazi, s strani več aplikacijskih strežnikov, bi namreč privedlo do napak v delovanju ciljne aplikacije in okvare podatkov.

5.8.1 Nevarne zahteve

Diffy privzeto senči samo varne spletne zahteve. Omogoča sicer tudi način, v katerem se senčijo zahteve z metodami protokola HTTP `POST`, `DELETE` in `PUT`, vendar pa ne nudi mehanizmov za upravljanje podatkovnih baz, ki so potrebni za preprečevanje okvare produkcijskih podatkov (replikacije produkcijske podatkovne baze v senčeno okolje). Kot rečeno, Diffy predvideva uporabo dveh produkcijskih aplikacijskih strežnikov, ki uporabljata isto podatkovno bazo. Posledično lahko pride do sočasnih pisalnih dostopov do istih podatkov produkcijske podatkovne baze s strani obeh aplikacijskih strežnikov, s tem pa potencialno do okvare produkcijskih podatkov in do napak v delovanju ciljne spletne aplikacije, ki jih občuti končni uporabnik. Velika pomanjkljivost rešitve Diffy v primerjavi z našo rešitvijo je torej v tem, da mehanizme za upravljanje podatkovnih baz v obeh aplikacijskih okoljih, potrebnih za pravilno in transparentno delovanje v primeru senčenja nevarnih zahtev, v celoti prepuščajo razvijalcu.

Nasprotno pa naša rešitev omogoča tudi transparentno senčenje nevarnih spletnih zahtev brez neželenega vpliva na produkcijsko aplikacijsko okolje. V naši rešitvi namreč posamezno zahtevo obdela en sam produkcijski aplikacijski strežnik. Prav tako v naši rešitvi en ali več senčenih aplikacijskih strežnikov uporablja repliko produkcijske podatkovne baze in ne produkcijske baze neposredno. Posledično obdelava nevarnih zahtev v senčenem okolju ne vpliva na delovanje aplikacije v produkcijskem okolju, zaradi česar je naša rešitev povsem transparentna z vidika končnega uporabnika. Upravljanje replikacije produkcijske podatkovne baze v senčeno okolje pa je v celoti integrirano s posredniškim strežnikom za senčenje, kar omogoča avtomatsko krmiljenje replikacije.

Poglavje 6

Sklep

V magistrskem delu nam je uspelo doseči vse zastavljene cilje: implementirali smo rešitev za avtomatsko zaznavanje vsebinskih in zmogljivostnih regresij, ki temelji na senčenju produkcijskih zahtev, in jo uspešno vključili v cevovod zvezne postavitve izbrane spletne aplikacije. Rezultat našega dela je celovita rešitev, ki obsega posredniški strežnik za senčenje spletnih zahtev, spletno aplikacijo za analizo in nadzor senčenja, knjižnico za sporočanje aplikacijskih metrik in mehanizem za upravljanje replikacije produkcijske podatkovne baze. Posredniški strežnik za senčenje produkcijskih spletnih zahtev omogoča transparentno podvajanje produkcijskih spletnih zahtev in njihovo obdelavo tako v produkcijskem, kot tudi na novo uvedenem senčenem okolju. Primerjava spletnih odgovorov obeh aplikacijskih okolij s strani spletne aplikacije za analizo in nadzor senčenja omogoča avtomatsko zaznavo tako vsebinskih kot tudi zmogljivostnih regresij novih izdaj, brez potrebe po časovno potratnem pisanju testov. Za razliko od edine nam znane alternative, Diffy, ki podpira samo senčenje varnih spletnih zahtev, naša rešitev omogoča tudi senčenje nevarnih zahtev. Nove izdaje lahko zato testiramo bolj celovito, saj nismo omejeni zgolj na peščico funkcionalnosti ciljne aplikacije.

Razvito rešitev smo integrirali z izbrano spletno aplikacijo, ki je že imela implementiran cevovod zvezne postavitve, in jo ovrednotili s pomočjo izbranih kriterijev. Ugotovili smo, da dodaten korak postavitvenega cevovoda, ki ga zahteva uvedba naše rešitve za senčenje zahtev, za izvajanje ne zahteva veliko dodatnega časa in infrastrukture. Uporaba realnih produkcijskih zahtev v postopku testiranja je omogočila analizo veliko večjega števila robnih pogojev in načinov uporabe ciljne aplikacije kot obstoječi načini avtomatskega testiranja v cevovodu zvezne postavitve, zaradi česar smo pridobili višjo stopnjo zaupanja v pravilnost novih izdaj.

Razvito rešitev je mogoče enostavno integrirati s spletnimi aplikacijami, ki uporabljajo ogrodje Ruby on Rails in sistem za upravljanje podatkovnih baz PostgreSQL. Naša rešitev

lahko prav tako služi kot izhodišče za implementacijo senčenja produkcijskih zahtev spletnih aplikacij v poljubnem programskem jeziku, ki uporabljajo katero drugo ogrodje za izdelavo spletnih aplikacij ali kateri drug sistem za upravljanje podatkovnih baz. V tem primeru integracija od razvijalca spletne aplikacije zahteva nekoliko več truda, saj zahteva bodisi drugačne prilagoditve v spletni aplikaciji in zamenjavo knjižnice za sporočanje aplikacijskih metrik bodisi prilagoditev ukazov za upravljanje replikacije, ki jih uporablja posredniški strežnik za senčenje.

Naša rešitev za avtomatsko zaznavanje regresij s pomočjo senčenja ima določene omejitve, ki so predvsem posledica kompromisov, sprejetih tekom implementacije. Na tem mestu želimo izpostaviti predvsem natančnost analize zmogljivostnih metrik senčenih zahtev. Implementirani postopek analize je namreč zelo preprost in ima omejeno natančnost v primeru večjih nihanj zmogljivostnih metrik. Za bolj verodostojno zaznavanje zmogljivostnih regresij bi bilo potrebno zbiranje večjega števila zmogljivostnih metrik v daljšem časovnem obdobju, ki bi jih kasneje podrobneje analizirali.

Za pravilno zaznavanje vsebinskih regresij moramo v ciljni spletni aplikaciji prilagoditi privzeta pravila za izločanje šuma. Prav tako morajo razvijalci za vsako pričakovano spremembo v spletnih odgovorih na dano zahtevo vnesti pripadajoče pravilo, kar v praksi traja nekaj minut. Kljub temu naša rešitev na področju testiranja novih izdaj spletnih aplikacij predstavlja veliko izboljšanje, saj alternativni pristopi zaznavanja regresij s produkcijskimi podatki, kot so modro-zelene postavitve in kanarske izdaje, za razliko od naše rešitve zahtevajo dolgotrajno ročno preverjanje pravilnega delovanja nove izdaje s strani razvijalcev, ki lahko traja več ur.

Čeprav je integracija naše rešitve s postavitvenimi cevovodi spletnih aplikacij, ki uporabljajo ogrodje Ruby on Rails, relativno enostavna, zahteva previdno konfiguracijo. Le s premišljeno izbiro vrednosti določenih parametrov namreč lahko dosežemo učinkovito in karseda natančno zaznavanje regresij. Ti parametri, na primer delež senčenih spletnih zahtev in mejna vrednost števca nevarnih zahtev, so odvisni predvsem od obremenitve ciljne aplikacije v produkcijskem okolju. Določanje ravno pravšnjih vrednosti omenjenih parametrov v praksi zahteva nekaj poskušanja.

Zavedamo se torej številnih možnosti za nadaljnje delo in izboljšave. Kljub vsem omenjenim pomanjkljivostim pa naše delo predstavlja prvo nam znano celovito rešitev za odkrivanje regresij v spletnih aplikacijah s pomočjo senčenja produkcijskih zahtev. Čeprav senčenje produkcijskih zahtev danes v praksi uporabljajo zgolj velika tehnološka podjetja, menimo, da se bo v naslednjih letih njegova uporaba razširila. Prednosti, ki jih prinaša avtomatsko zaznavanje regresij s pomočjo senčenja produkcijskih zahtev, namreč najbolj pridejo do izraza predvsem za spletne aplikacije s pomanjkljivim avtomatskim testiranjem, slednje pa predstavljajo velik delež vseh spletnih aplikacij.

Literatura

[1] B. Adams, S. Bellomo, C. Bird, T. Marshall-Keim, F. Khomh in K. Moir, "The practice and future of release engineering: A roundtable with three release engineers", *IEEE Software*, št. 32, zv. 2, str. 42–49, 2015.

[2] (2016) Bucardo. Dostopno na: https://bucardo.org/wiki/Bucardo (pridobljeno 28.11.2016).

[3] L. Chen, "Continuous delivery: Huge benefits, but challenges too", *IEEE Software*, št. 32, zv. 2, str. 50–54, 2015.

[4] M.-H. Chen, M. R. Lyu in W. E. Wong, "Incorporating code coverage in the reliability estimation for fault-tolerant software", v zborniku *Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on.* IEEE, 1997, str. 45–52.

[5] (2016) Continuous integration, deployment & delivery with codeship. Dostopno na: http://codeship.com (pridobljeno 16.11.2016).

[6] (2016) Deploying the Netflix API. Dostopno na: http://techblog.netflix.com/2013/08/deploying-netflix-api.html (pridobljeno 16.11.2016).

[7] (2016) Diffy: Testing services without writing tests. Dostopno na: https://blog.twitter.com/engineering/en_us/a/2015/diffy-testing-services-without-writing-tests.html (pridobljeno 28.11.2016).

[8] E. W. Dijkstra, "The humble programmer", *Communications of the ACM*, št. 15, zv. 10, str. 859–866, 1972.

[9] T. Dingsøyr in C. Lassenius, "Emerging themes in agile software development: Introduction to the special section on continuous value delivery", *Information and Software Technology*, št. 77, str. 56–60, 2016.

[10] (2016) Eventmachine: fast, simple event-processing library for ruby programs. Dostopno na: https://github.com/eventmachine/eventmachine (pridobljeno 28.11.2016).

[11] (2016) Eventmachine proxy dsl for writing high-performance transparent / intercepting proxies in ruby. Dostopno na: https://github.com/igrigorik/em-proxy (pridobljeno 16.11.2016).

[12] (2016) Experella-Proxy - a ruby reverse proxy using eventmachine. Dostopno na: https://github.com/experteer/experella-proxy (pridobljeno 16.11.2016).

[13] D. G. Feitelson, E. Frachtenberg in K. L. Beck, "Development and deployment at Facebook." *IEEE Internet Computing*, št. 17, zv. 4, str. 8–17, 2013.

[14] (2016) Github rack/rack: a modular ruby webserver interface. Dostopno na: https://github.com/rack/rack (pridobljeno 28.11.2016).

[15] (2016) Github samg/diffy: Easy diffing in ruby. Dostopno na: https://github.com/samg/diffy (pridobljeno 28.11.2016).

[16] (2016) Hammering usernames. Dostopno na: https://www.facebook.com/notes/facebook-engineering/hammering-usernames/96390263919/ (pridobljeno 16.11.2016).

[17] (2016) How we build code at Netflix. Dostopno na: http://techblog.netflix.com/2016/03/how-we-build-code-at-netflix.html (pridobljeno 16.11.2016).

[18] J. Humble in D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[19] J. Humble, C. Read in D. North, "The deployment production line." v zborniku *AGILE*, št. 6, 2006, str. 113–118.

[20] (2014) Hypertext transfer protocol (http/1.1): Semantics and content. Dostopno na: https://tools.ietf.org/html/rfc7231#section-4.2.1 (pridobljeno 16.11.2016).

[21] (2016) Kage (kah-geh) is a shadow proxy server to duplex http requests. Dostopno na: https://github.com/cookpad/kage (pridobljeno 16.11.2016).

[22] E. Laukkanen, J. Itkonen in C. Lassenius, "Problems, causes and solutions when adopting continuous delivery—a systematic literature review", *Information and Software Technology*, št. 82, str. 55–79, 2017.

[23] S. Mäkinen, M. Leppänen, T. Kilamo, A.-L. Mattila, E. Laukkanen, M. Pagels in T. Männistö, "Improving the delivery cycle: A multiple-case study of the toolchains in finnish software intensive enterprises", *Information and Software Technology*, št. 80, str. 175–194, 2016.

[24] S. McConnell, "Daily build and smoke test", *IEEE software*, št. 13, zv. 4, str. 144, 1996.

[25] D. L. Mills, "Measured performance of the network time protocol in the internet system", Tech. Rep., 1989.

[26] G. J. Myers, C. Sandler in T. Badgett, *The art of software testing.* John Wiley & Sons, 2011.

[27] S. Neely in S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)", v zborniku *Agile Conference (AGILE), 2013.* IEEE, 2013, str. 121–128.

[28] (2016) New relic: Digital performance monitoring and management. Dostopno na: https://newrelic.com (pridobljeno 28.11.2016).

[29] (2016) Nginx. Dostopno na: https://nginx.com (pridobljeno 28.11.2016).

[30] D. Nir, S. Tyszberowicz in A. Yehudai, "Locating regression bugs", v zborniku *Haifa Verification Conference.* Springer, 2007, str. 218–234.

[31] H. H. Olsson, H. Alahyari in J. Bosch, "Climbing the "Stairway to Heaven" –A mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software", v zborniku *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on.* IEEE, 2012, str. 392–399.

[32] (2016) Pure ruby implementation of an ssh (protocol 2) client. Dostopno na: https://github.com/net-ssh/net-ssh (pridobljeno 16.11.2016).

[33] G. Schermann, J. Cito, P. Leitner in H. C. Gall, "Towards quality gates in continuous delivery and deployment", v zborniku *2016 IEEE 24th International Conference on Program Comprehension (ICPC).* IEEE, 2016, str. 1–4.

[34] (2016) Servertimingmiddleware. Dostopno na: https://github.com/javierhonduco/server_timing_middleware (pridobljeno 28.11.2016).

[35] M. Siddiqui, P. Commons, I. Gonzalez, A. Vance, K. Yourtee in T. Keller, "Software testing using shadow requests", Jun. 16 2015, uS Patent 9,058,428. Dostopno na: https://www.google.com/patents/US9058428

[36] (2016) Skylight. Dostopno na: https://www.skylight.io/ (pridobljeno 28.11.2016).

[37] (2016) Slony-i. Dostopno na: https://slony.info (pridobljeno 28.11.2016).

[38] (2016) Streaming replication. Dostopno na: https://wiki.postgresql.org/wiki/Streaming_Replication (pridobljeno 28.11.2016).

[39] (2016) The recipe for the world's largest Rails monolith. Dostopno na: https://speakerdeck.com/a_matsuda/the-recipe-for-the-worlds-largest-rails-monolith (pridobljeno 16.11.2016).

[40] (2016) Velocity Culture. Dostopno na: http://assets.en.oreilly.com/1/event/60/Velocity%20Culture%20Presentation.pdf (pridobljeno 16.11.2016).